A Standards-Based Approach to Dynamic Tool Integration Using Java Business Integration

A Redesign of the ToolNet Framework built on Enterprise Integration Standards

Gregor B. Rosenauer

A Standards-Based Approach to Dynamic Tool Integration Using Java Business Integration: A Redesign of the ToolNet Framework built on Enterprise Integration Standards

Gregor B. Rosenauer
Supervisor: Ao.Univ.Prof. Dr. Stefan Biffl
ISIS Vienna University of Technology Institute of Software Technology and Interactive Systems
Co-Supervisor: Univ.Ass. Dr. Alexander Schatten
ISIS Vienna University of Technology Institute of Software Technology and Interactive Systems

Published October 2008

To Janna, my loving wife

To all who continue to strive for open standards interoperability

Table of Contents

List of Figures

List of Tables

List of Examples

Kurzfassung

Die gegenseitige Integration von heterogenen Tools mit dem Ziel, den Arbeitsablauf von Benutzern zu optimieren, ist Gegenstand andauernder Forschung. Die angestrebte Lösung soll Benutzern und Teams ermöglichen, bestehende Tools auf transparente Art miteinander zu verbinden. Funktionalität und Daten von einzelnen Tools können von jedem anderen Tool aus verwendet werden; Gemeinsamkeiten im Datenmodell werden ausgenützt, indem man Relationen zwischen zusammengehörenden Datenelementen erzeugt.

Eine besondere Herausforderung stellt die flexible Integration von bestehenden, meist kommerziellen Tools dar, wie sie z.B. im Ingenieurswesen vorkommen. Diese bieten oft nur proprietäre und nicht offen zugängliche Schnittstellen an, was das Design einer Integrationslösung in vielerlei Hinsicht einschränkt. Es wurden bereits verschiedene Frameworks und Standards entwickelt, wie z.B. CDIF, PCTE, OTIF, BOOST oder auch allgemeine Tool-Plattformen wie z.B. Eclipse. Diese lösen aber jeweils nur einen Teil des Problems und bieten keinen ganzheitlichen, dynamischen Ansatz für die Integration von bestehenden bzw. proprietären Tools.

Eine erfolgreiche Lösung für die Tool-Integration muss die Anforderungen verschiedener Gruppen gleichermaßen erfüllen: Für den *Endbenutzer* steht eine nahtlose Integration zwischen Tools im Vordergrund, die es ermöglicht, transparent über Tool-Grenzen hinweg zu arbeiten. *Entwickler* wünschen sich einen einfachen Weg, um Tools in ein lose gekoppeltes und dynamisches System einzubinden, das leicht um neue Tools erweiterbar und an geänderte Schnittstellen anpassbar ist. *Toolhersteller* wollen unabhängig bleiben und zusätzliche Kosten für die Neuimplementierung oder Anpassung von Tools an Integrationslösungen vermeiden, bieten aber als Ausgleich oft Skripting- oder sprachspezifische Schnittstellen für die Anbindung an andere Anwendungen an, die man für die Tool-Integration nützen kann.

Diese Arbeit zeigt, dass die Tool-Integration am Desktop viel mit der Enterprise Integration gemeinsam hat, wo es bereits eine Reihe von "best practices", Mustern und Integrations-Standards wie z.B. *Java Business Integration* (JBI) oder die *Service Component Architecture* (SCA) gibt. Die Anwendung erfolgreich erprobter Lösungen aus der Enterprise-Integration auf die Tool-Integration am Desktop ermöglicht die Umsetzung einer wiederverwendbaren und erweiterungsfähigen Integrationslösung, die leicht an neue Tools und Anforderungen angepasst werden kann. Aufbauend auf einer Analyse der aktuellen Situation wird unter Verwendung von JBI ein standardbasiertes dynamisches Framework für die Tool-Integration realisiert.

Eines der wenigen existierenden Frameworks, die eine solche Integrationslösung umsetzen, ist ToolNet, ein von der EADS CRC Deutschland entwickeltes serviceorientiertes Framework für die Tool-Integration. ToolNet verbindet existierende kommerzielle Tools aus dem Ingenieursbereich – wie z.B. Telelogic DOORS oder Matlab – mit Hilfe von speziell entwickelten Adaptern, die über einen gemeinsamen Nachrichtenbus kommunizieren. Ausgehend von einer Analyse der Ist-Architektur und ihren Einschränkungen, die vor allem in der statischen und proprietären Adapter-Architektur bestehen, werden die Forschungsergebnisse dieser Arbeit in einem Prototypen umgesetzt, der ein Redesign der ToolNet-Architektur basierend auf dem JBI-Standard und einem dynamischen Adapterkonzept demonstriert. Der Prototyp wird danach einer Evaluierung unterzogen und mit dem bestehenden ToolNet-Framework verglichen.

Abstract

Integrating heterogeneous software tools with each other on a peer-to-peer level for streamlining the end user's workflow is an area of ongoing research. The ideal tool integration solution would provide users a transparent way to integrate and connect existing tools, without leaving the native interface. Functionality of individual tools can then be shared and commonality in data models is exploited by creating relations between corresponding data elements.

A special problem is the flexible integration of existing, often commercial-off-the-shelf (COTS-)tools, as encountered e.g. in the engineering domain. These often provide only proprietary and closed APIs with limited capabilities, posing various restrictions on the design of a prospective integration solution. Several frameworks and standards, such as CDIF, PCTE, OTIF or BOOST have been developed, including general-purpose tool platforms like Eclipse, but so far these have only solved parts of the problem, lacking a holistic, dynamic approach for integrating existing or proprietary tools.

From a user's perspective, tight integration between tools is desired, facilitating working across tool borders in a transparent way. From a developer's perspective, loosely-coupled integration and a dynamic way to integrate new tools into the framework with little effort is desired, sothat the resulting solution is easily adaptable to new tools and changing APIs. Tool vendors want to stay independent and will not accept additional cost for reimplementing or adapting tools to work with specific integration solutions, but often provide scripting interfaces and language-specific APIs for connecting tools to other applications.

This work demonstrates that tool integration faces many of the same challenges encountered in enterprise integration, where already several best practices, patterns and integration-standards such as *Java Business Integration* (JBI) and the Service Component Architecture (SCA) have evolved. By applying successful solutions from enterprise integration to the problem of tool integration on the desktop, a reusable and extensible integration solution can be realized that is easily adaptable to new tools and requirements. This work examines the current situation and demonstrates howe the JBI standard can be utilized for tool integration, propsing a standards based, dynamic tool integration framework.

One of the few existing tool integration solutions that target this problem is ToolNet, a custom, service-oriented integration framework developed by EADS Corporate Research Centre Germany. ToolNet connects existing, commercial off-the-shelf engineering tools, such as Telelogic DOORS or Matlab, using custom Adapters and a proprietary messaging backbone. After an analysis of the current architecture and its limitations, mainly the static Adapter architecture, the findings in this work are applied in a prototype implementation that demonstrates a redesign of ToolNet based on the JBI standard. The prototype is then evaluated and compared to the existing ToolNet framework.

Preface

This thesis is the result of over 2 years of work – with certain distractions such as civil service, a full-time job and finally my own wedding:) – on researching integration approaches on the desktop and in the enterprise world. The work is both a theoretical survey on the diverse aspects of integration in order to find new solutions for tool integration that allow keeping tools as-is, and at the same time it is also a practical work that was initiated by EADS Corporate Research, Germany. In this part, the findings gained through the theroretical analysis are subsequently applied and evaluated in a prototype implementation, redesigning an existing tool integration framework, ToolNet, developed by EADS. This dual approach has led to a comprehensive and relatively mature, but hopefully insightful and approachable result, building on practical experiences reflected against a solid theoretical background. I hope it provides some new insights to a long-standing research problem, and sparks interest in the field, as well as motivating the reader to end the struggle against isolated, incompatible applications, data formats and cumbersome work“flows”.

When I started initial research on this work in June 2006, little did I know about how diverse and broad the field of integration is, spawning several dimensions and layers, and how many integration projects, both commercial and academic, and related standards efforts have been undertaken in the last years. It both astounding and regrettable that after all these years, only very few solutions have emerged that are currently available for integrating tools on end user's desktops. With the exception of a few specialized commercial offerings, and well-known but isolated and vendor-specific tool suites, users still have to copy&paste information between applications or tediously export and import files, because common operating systems only provide low-level data-integration as opposed to semantic or more service-oriented integration (with the notable exception of Apple MacOS which provides some user-oriented integration services and an interface through Automator). Even on the data level, only recently office documents have been standardized (even twice!) to allow exchange between different office applications.

The lack of suitable, cross-platform integration standards on the desktop led me to investigating enterprise integration more closely. Because the ToolNet framework is Java-based, a Java-based solution or standard was preferred. Also, the Java world is traditionally more open than other platforms, looking at Apache, JBoss or recently even traditional industry heavyweights like IBM (Eclipse) and Sun (Open*). During my search for truely open and multi-platform integration possibilities, struggling to keep sane in the SOA jungle, I found a promising solution: Java Business Integration (JBI, JSR-208).

At that time, JBI was still a very young standard, largely unknown in research (common search engines yielded zero results) and in the developer community. As a result, it was challenging to find related information besides blogs, wikis or sample code. Following the emerging developer landscape required constant re-evaluation, further research and trying out many different solutions and concepts. The arrival of the Service Component Architecture (SCA)-standard in March 2007 did not make things any easier, as now there were two closely related standards to evaluate and differentiate, which was not easy even for experts in the field, and resulted in heated discussions on the web.

The Web was another source for inspiration – the proliferation of recent Web 2.0 mashups that are in the hands of users, connecting disparate applications in a spontaneous and unpredictable way that is out of control of their originators, has shown the huge mutual potential fo users, developers and companies. Every day, new solutions are formed out of existing, autonomous applications. These composite applications are more than the sum of their parts, and motivate a similar approach on the desktop.

So for a successful tool integration approach, we should investigate all three major application domains – the enterprise, the web and the desktop – as there is a significant overlap in all of them, and a similar need for integration. From enterprise integration, we can take many patterns and solutions that have evolved over the years and applied in large scale integration projects. From the web, we gain more user-oriented, spontaneous and dynamic integration concepts and interfaces. The desktop brings highly specialized and rich tools that can

be used anywhere, online or offline, and that scale from mundane tasks like spreadsheets to highly demanding tasks like video editing or CAD.

The resulting solution using JBI was challenging to apply in the ToolNet redesign, as ToolNet comprises a huge codebase, and the API is very complex. Because the prototype should stay independent but at the same time offer a migration path, only the original tool-side scripts (used to integrate Telelogic DOORS) were reused and connected to a new implementation built around a JBI based ESB, Apache ServiceMix, which was still in incubation when the prototype was started.

During the last years, there was a major shift towards standards in integration design (e.g., with SCA) and implementation, with JBI proposing a common runtime infrastructure that facilitates open enterprise service busses (ESBs) and reuse of composite applications across implementations unlike current, often vendor-specific implementations.

It is time to investigate how tool integration can profit from these advances, as making tools talk to each other still remains a major challenge. This thesis strives to provide a starting point for a new generation of tool integration frameworks, embracing open standards from Enterprise and Web 2.0, and bringing them to the desktop for the benefit of end users.

Acknowledgements

Several parties and people were involved in this thesis and supported me on various levels:

I want to thank my supervisors Alexander Schatten and Stefan Biffl from the Vienna University of Technology, IFS group, for providing a very exciting research topic combined with a practical project, and contacts to the industry.

Many thanks to Andreas Keis and Martin Klaus from EADS Corporate Research Germany, for inviting me to Munich and providing me some insight into a complex real-world integration framework, ToolNet. Especially the subsequent technical Skype-sessions with you, Martin, were very helpful – they directed me to the relevant parts of the framework, enlightened me on several concepts behind ToolNet and cleared up some open issues. They also helped me in shaping my proposed solution and evaluating my design ideas.

I also want to express my gratitude to my friends Manfred Jakesch and Martin Thelian for valuable and constructive feedback on this work as it progressed, and generally for supporting me during this long period with positive encouragement.

Finally, I can hardly describe the deep support and motivation my wife, Janna, has given me throughout my work, despite writing her own thesis. To a degree, it is true that “A problem shared is a problem halved.”, but it has been a busy time for sure.

Chapter 1. Introduction

Like a bridge over troubled water I will lay me down.
--Paul Simon, "Bridge Over Troubled Water"

1.1. Overview

The integration of software tools, esp. *commercial off-the-shelf* (COTS)-tools, is a long-standing need not only in the enterprise and server-side world, but also on the *client side*, where usually a mix of isolated pre-packaged tools is used, as for example in the engineering domain. The goal of an integrated workflow is hindered by manifold limitations on the functional level (such as restricted APIs, missing data exchange functionality), on the presentation level (so the user interface cannot be accessed by external tools) and on the data level (e.g., incompatible, closed or legacy formats). Current market offers provide mostly custom, commercial frameworks like [BizTalk] or [OpenSpan] that bear the danger of vendor lock-in and limit users to specific integration solutions and proprietary platforms. Only recently, more open, standards-based approaches like XAware [XAware] or Xcalia [Xcalia] emerged with open source implementations for data and service integration, respectively, by using open standards (such as *Service Data Objects*, SDO, see Section 8.2, and SCA or JBI (see below), which will be explained in Chapter 4).

Although the problem of integrating applications has been identified earlier (see Chapter 2), current developments only target the enterprise domain, successfully integrating business processes and legacy applications, as examined in [Microsoft2004], which has resulted in several best practices and patterns being available to enterprise integration architects, such as [EIP] and [PoEAA]. Very few of these solutions have been applied to *client-side* or *desktop integration*. For example, [Balasubramanian2006] proposes a formal concept of *model-driven integration* for integrating COTS products in the enterprise, which leads to a clean, high-level functional integration but is impractical for dynamically combining COTS products, where the deduction of models is often hindered by missing information on the internal architecture and functionality. [Damm2000] applies a model driven approach in the Knight whiteboard tool, using *XMI* (XML metadata interchange) for data integration, but relies on *COM* for communicating with COTS modeling tools, which creates a tight coupling between integrated applications. Model based approaches also require specialized tools and thorough modeling knowledge for designing the integrated meta-model.

The solution presented in this work approaches the problem domain of *tool integration*, including *COTS* tools, using *service-oriented integration* (*SOI*, Section 3.3.3)-approach combined with *message-based integration*, examining existing and emerging specifications such as *Java Business Integration* (*JBI*) and the *Service Component Architecture* (*SCA*). It shows how current integration solutions could benefit from applying recent developments in the enterprise to the desktop, using an industrial integration framework, *ToolNet*, as a case study. The *ToolNet*-framework [Mauritz2005] was developed by EADS CRC Germany to fill this need by integrating various applications used in the aeronautic engineering domain. The framework is loosely based on the OSGi component framework and uses the *Eclipse Rich Client Platform* (*Eclipse RCP*) to provide a simple user interface for managing integrated tools.

ToolNet connects legacy applications through the use of Adapters (see Section 5.2.2), providing a way for combining previously isolated tools into an integrated tool chain. Users can then define *Relations* for integrating related data elements in individual tool models. By transforming legacy APIs into services available to any participant on the framework's backbone, the original COTS tools can even be extended with new functionality such as distributed collaboration or additional data formats. Through `Relations`, automatic workflows can be realized which previously required manual steps. Connections between the integrated applications are live, so changes are propagated between connected tools integrated in the *ToolNet*-infrastructure. This solution has been successfully used to integrate COTS tools such as Telelogic *DOORS* (a requirements-tracing tool), Microsoft Word or Matlab.

A more detailed analysis of the framework (provided in Chapter 5) has shown a mixed and more or less static architecture that complicates integration of new tools and adaption to new versions of already integrated tools. Also, custom solutions are used where already industry standards are available, but their adoption is hindered by architectural constraints. Lastly, the user-interface of the management console (*ToolNet* Desktop) is limited to local control of *Tool Adapters* and the definition of tool relations, but offers no lifecycle-management or advanced remote monitoring, which is crucial for a flexible and reliable integration solution.

In this thesis, *Java Business Integration* (JBI) and related integration concepts are applied in an architectural redesign of ToolNet, which is then implemented as a prototype. As with the original solution, the new solution is then evaluated based on the requirements identified earlier, and finally compared to the current implementation of ToolNet.

Lastly, current developments in the field, such as the recently started *JBI 2.0* specification ([*JSR 312*]), are covered together with a look into the future of service-oriented tool-integration and related emerging architectures.

1.2. Related Work

The integration of software tools to foster interoperability and communication among (mostly software) engineering teams has been an ongoing research topic since the late 1980s, when the problem was described in [Wasserman1989], which defined several levels of integration in software engineering environments (then called CASE tools), a definition which is still used today, in an extended form.

An excellent and extensive literature overview is given in [Wicks2006] and [Wicks2007]. The latter performs a critical evaluation on tool integration-research on a meta-level, posing the question if the right problems have been targeted or whether research is going in the wrong direction. As a result, a new research agenda for solving the remaining problems is proposed, suggesting a more market-oriented approach that targets real world-problems and business requirements like increased return of investment (ROI).

The problem of COTS integration has been identified early as an important factor in tool integration, e.g., in [BaoHorowitz1996], which evaluates the BOOST project, an EU initiative to create an open framework for integrating existing, closed source tools in engineering processes. [Warboys2005] investigates deployment and lifecycle issues and proposes an adaptive architecture that is capable of handling dynamically changing software installations which include closed COTS products. Chapter 3 describes more closely related solutions that focus on the integration of independent but related tools in a more transparent way, and concentrates on the problem of how tools interact with each other and how the integrated system can interact with end users.

On a more general level, several standards for tool integration in engineering have been proposed. An early example is the *Portable Common Tool Environment* (PCTE), which defined an open repository for tools and acted as a shared database, providing various language bindings for adapting existing tools to connect to the PCTE. [Anderson1993] gives a good overview and performs an evaluation the framework. PCTE was later adopted as a standard by the ECMA (ECMA-149). There was also an ANSI standard[1], which is mentioned in early literature, but not available anymore. The Open Tool Integration Framework [OMG2004] “seeks to create a standard for an open tool integration framework that would support separating the tools to be integrated from the framework used to facilitate the integration.”. The suggested framework supports two scenarios:

1. *tool chains* (using *process integration*), where integrated tools are connected into a coherent workflow, where *Adapters* or *Translators* are used for bridging different APIs and file formats (XML, XMI, or proprietary formats)

2. *ad hoc data sharing* via repositories or meta models, which requires a significant overlap in data models, e.g., engineering tools from the same domain

[1] X3H6 Standard Committee, "Proposed Draft Standard Messaging Architecture", Document X3H6/93-012, July 1993

Although the request for proposal did not reach final approval, it provides a valuable foundation for standards based tool integration solutions and validates the approach used in similar frameworks presented below.

[IEEE2006] provides a reference model for tool interconnections, building on the CDIF (CASE Data Interchange Format)-standard introduced in [Parker1992], who proposes a common format that facilitates data integration among software engineering tools.

Communication in tool integration is often realized through messaging, which is a high-level form of *functional integration*. [Verrall1992] is an early example for CASE tool integration using a message bus, the "software bus", introducing the concept of *software factories* (c.f. [Greenfield2004]) implemented as a *Factory Support Environment*, which is defined as "a distributed communications-oriented CASE environment.". [Arnold1995] describes various methods of control integration and refers to the ANSI X3H6-standard, which tried to standardize various inter application communication protocols such as ToolTalk (see [Sun1993] and [Julienne1994]), one of the first approaches to provide an OS-level API for tool integration (see Section 3.2.1), as well as CORBA and similar distributed object models.

[Guo2004] is an example for a *component-based* approach that integrates tools using a *canonical* interface (specified in IDL) and a communication backbone, modeled as a message bus, the ToolBus, using CORBA (see Section 2.6.1 for a discussion on the problems with CORBA-based approaches, including firewalls, the inherent performance penalty, tight coupling and bad mapping to modern programming languages like Java). Later, similar frameworks based on Web services emerged (a recent example is ToolNet which is covered in the case study in Chapter 5. Web-service based integration will be discussed in Section 3.3.3.1.

[Balasubramanian2006] proposes an approach using model driven integration (which is covered in Section 2.3.7 and Section 3.3.6), using a *generic modeling environment* (GME), where integration architects describe an integration problem at a high level using a domain-specific language (DSL), the *System Integration Modeling Language* (SIML). Integration is then done at the *functional* level, a concept detailed in Section 2.3.4. The work suggests that integration has to be handled at a higher level, following that "attempting integration at the wrong level of abstraction can yield brittle integration architectures that require changes to the integration architecture when changes occur to either the source or target system being integrated" [Balasubramanian2006:7]. This demonstrates a major requirement of integration solutions: *tight coupling*, as existent in low-level integration approaches, must be avoided in favor of *loosely coupled* integration, which reduces inter-dependencies between integrated systems.

[Corradini2004] proposes an *agent-based* approach, which is well suited for *process integration* and supporting users in data mining, transparently accessing different tools to gather the required information. The suggested solution uses information integration, building ontologies of the target domain with the use of autonomous agents that coordinate each other through messaging.

Integration frameworks like the previously mentioned Knight tool environemnt and ToolNet have tried to solve these problems in different ways, the former using a model based approach based on COM and XMI, the latter using a mix of custom solutions, service oriented concepts and a customized, OSGi-based plugin framework (see Chapter 5 for a detailed analysis of ToolNet and Section 3.2.3.1 for more on OSGi).

A more standards based approach is presented in [Yap2005], where a framework for extending applications with *web services* is presented at the example of the free Java editor jEdit. Web services may also be used as wrappers for legacy services, as shown in [Sneed2005], targeting enterprise integration.

Only recently, high-level integration standards like JBI and SCA have been applied in research to solve enterprise integration problems, such as [Chen2007] who provides a distributed JBI environment with additional tool support for integrating existing, isolated systems with proprietary interfaces, and [Ning2008], who examines distributed JBI using JMS (as JBI currently does not specify distributed environments). [Ruiz2008] applies the related SCA-standard for developing a service-oriented electronic banking architecture as part of the Spanish

ITECBAN project adding support for missing SCA functionality such as distributed deployments and Service versioning.

The solutions presented here demonstrate the need for standards-based application integration frameworks, but at the same time they show the challenges caused by limited proliferation of standards suitable for tool integration frameworks and the lack of common APIs that provide integration architects with a high-level solution to the diverse scope of integration problems. Only recently, such integration architectures have emerged, inspired by successful solutions in the enterprise domain, but they have not yet been applied to the desktop domain. Chapter 3 surveys current integration concepts and available standards on the desktop in more detail, drawing analogies to related solutions in enterprise integration. Section 2.5 provides a short review on the evolution of tool integration solutions on the desktop.

1.3. Target Audience

The topic of integration will be of natural interest to integration architects, system designers and software architects in the enterprise application domain. Chapter 3 provides some insight into COTS integration, whereas Chapter 4 covers state-of-the-art integration architectures and patterns, especially but not limited to the Java-space, namely JBI and SCA. Chapter 6 serves as a practical example of how JBI together with an *enterprise service bus* (ESB) can be successfully applied to a concrete integration problem where closed source legacy applications have to be integrated. As a case study for refactoring an existing integration solution, a look at Chapter 5 and Chapter 4 is recommended.

Application developers will be mostly interested in how they can provide access to their own creation to outside developers in an easy way which reaches beyond proprietary APIs or scripting interfaces. This is explained in Chapter 4; for Java-developers, Section 4.2 might be of special interest, as well as Chapter 6, which describes how to add monitoring and management access to Java applications using the *Java Management Extensions* (*JMX*) API. The chapter also provides some general analysis and design insight into service-oriented development with Java.

SOA developers will want to look at Chapter 3 for an analysis on current web-service developments targeted at integration, which also shows the limits of a *web service*-only approach. Section 3.3.5 shortly introduces event-driven architectures (*EDA*), a related but complementary approach that acts more indirectly and could prove a flexible alternative to purely service-oriented architectures in various scenarios.

Lastly, system administrators will be interested in the desktop integration of COTS software, which is introduced in Section 3.2 and detailed in Chapter 3. Also, the possibilities of remote administration and system control using *JMX* might be of interest, which is covered in Section 6.4.3.5.4.

1.4. Chapter Overview

The following chapter, Chapter 2, gives a more detailed overview of the problem domain together with background information on desktop integration of COTS tools, the main focus of this work. The chapter also provides the necessary context to enterprise integration, spanning from past integration concepts and failures in the enterprise (*EAI*) to the recent development of integration patterns and best practices, and finally hints at the current move towards integration frameworks that combine several patterns and best practices to offer a complete solution for integration architects and developers.

Chapter 3 complements the previous chapter with an analysis on the current state of the art in desktop integration. Open standards and concepts such as *service oriented architecture* (SOA) and *event-driven architecture (EDA)* are introduced, leading to recent integration efforts and disciplines like *Business Process Modeling* (BPM), *service oriented integration (SOI)* and second-generation web standards (WS-*). The chapter covers concrete inte-

gration concepts on the desktop, from Tcl/Tk to AppleScript, up to recent frameworks like OSGi and Eclipse, and shows related approaches in enterprise integration, such as the *Enterprise Service Bus(ESB)*, *Web Service-integration* , and associated standards such as WSIF, *JCA*. Finally, complete solutions currently available on the desktop and in the enterprise are presented, both open-source and commercial.

In Chapter 4, the research question of integrating COTS applications based on recent developments in *standards-based integration* is presented, applying the findings outlined in the previous chapters. Suitable patterns and best practices are selected and adopted to solve the research problem. Challenges and key issues in desktop integration projects are examined, and the resulting requirements are outlined, to be later applied in the prototype, which is covered in detail in Chapter 6. The chapter includes a survey on modern open source Enterprise Service Bus (*ESB*)-implementations and evaluates the best match for the prototype. Finally, the proposed solution based on JBI and Apache ServiceMix is presented, where the two major industry efforts on service integration and standardization, JBI and SCA, are examined more closely.

Chapter 5 provides a detailed analysis of a current COTS integration solution as a case study. After describing the vision, motivation and the target domain of aeronautic engineering, the current design is examined, revealing its merits and drawbacks. Also, common use cases with successfully integrated COTS tools are shown, especially the DOORS application, which is used for requirements engineering and serves as a practical use case in the subsequent chapter. Special attention is being paid to the Adapter-architecture, as this is a key point in COTS integration, taking the DOORS Adapter as an example.

Chapter 6 is the practical counterpart to Chapter 4 and presents a redesign of the previously examined integration framework, ToolNet. The new approach resembles the *ToolNet*-vision but takes a different integration approach based on the *Java Business Integration*-specification and standards based integration, using an open source ESB implementation, Apache ServiceMix. The prototype makes use of enterprise integration-patterns and best-practices that have been found applicable to COTS integration in Chapter 4, such as mediated message exchange (see Section 2.3.4), service oriented integration (SOI) or the *Java Connector Architecture* (JCA), which is explained in more detail. As a practical prototype scenario analogous to the existing ToolNet setting, the aforementioned DOORS tool is integrated using the new Adapter architecture, serving as a proof-of-concept of the new integration approach. Some use cases are provided to illustrate the features and functionality of the new implementation.

Chapter 7 performs a critical evaluation of the new approach, including a comparison to the current ToolNet implementation, and a validation of the requirements identified earlier. Strengths and challenges of the proposed solution are discussed, followed by a short investigation on how the existing ToolNet implementation could be migrated step by step to the new architecture, allowing for a parallel operation by bridging the two solutions.

Chapter 8 provides a prospective view on future developments in the tool integration space, giving some insight into coming specifications and possibilities, such as *Java Business Integration* 2.0 or the SDO-standard, and emerging architectural paradigms such as *resource-oriented architecture* and domain-specific languages for integration. Here we also look at increasing Java-side scripting support and novel tools that strive to provide integration architects and end-users with powerful ways to design and experience next generation tool integration solutions.

Part I. Behind Integration: Challenges and Current Situation

Table of Contents

This part provides a theoretical background of integration and structures current concepts and approaches in several levels, which are later combined to form new integration approaches.

Chapter 2. Problem Definition: Integration Challenges

2.1. Motivation

We need techniques that allow us to take applications that were never designed to interoperate and break down the stovepipes so we can gain a greater benefit than the individual applications can offer us.
--Martin Fowler in his foreword to Enterprise Integration Patterns [EIP]

As the now classical book on enterprise integration, *Enterprise Integration Patterns* [EIP], puts it in the introduction: “interesting applications rarely live in isolation”, adding that “it seems that any application can be made better by integrating it with other applications”. Although targeted at the enterprise domain, where *legacy applications* are connected in ways that were not anticipated when they were originally developed, these statements are also true for the desktop, where there is an increasing demand to combine existing, often pre-packaged applications (subsequently termed *commercial-off-the-shelf (COTS) applications*) from different vendors in order to facilitate a streamlined workflow that adapts to the user's needs. This problem domain is still rather young and referred to as *desktop application integration*, where the aim lies in enabling users of desktop applications to step beyond isolated tools or single-vendor "suites" that try to offer a complete solution for specific markets (e.g., office productivity, creative design etc.), but bear the danger of vendor lock-in, towards dynamically customized *composite applications* tailored to the personal workflow and working style. Breaking the barriers imposed by closed and isolated desktop applications in heterogeneous system landscapes through modern integration approaches can help to enhance usability and productivity, and also reduces cost by reusing existing applications in ways that were not possible before.

The majority of integration approaches that have emerged so far mainly focus on the *enterprise* domain and address the problem of integrating existing isolated applications. There has been a lot of pressure from corporate decision makers to ensure that systems interoperate, for political, cost and performance reasons. It would be unacceptable if a business-critical backend system (e.g. a *CRM* system) could not work together with a common directory server that stores related business contacts. Consequently, during the last 10 years, a new discipline in software engineering has evolved, called *enterprise integration*: Although the enterprise had moved away from earlier, centralized and monolithic systems to more open and distributed applications, integration between individual applications was needed to support changing business requirements and processes. Early integration attempts resulted in monolithic and complex integration backbones that were either custom developed or vendor-specific. Only in recent years, vendor-neutral standards and more high-level approaches have been established, including *service-oriented integration* (see Section 3.3.3) or *event-driven integration* (see Section 3.3.5), solving the integration problems at a higher level and providing more flexibility and potential for reuse.

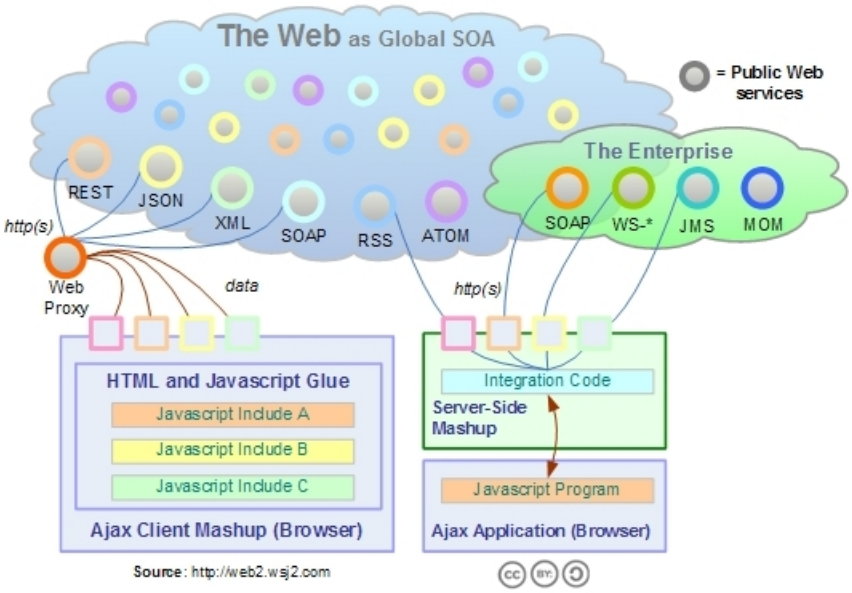
Several solutions have been developed for combining existing software assets in heterogeneous environments, in order to secure investments and to adjust to the needs of dynamically changing business processes (see Section 3.3.4 for a short coverage of *BPEL* and related business orchestration standards). The evolution of enterprise integration is covered in more detail in Section 2.6 at the end of the chapter. Based on the success of *design patterns* in software development, as introduced by [GoF] and later [POSA], also patterns for enterprise integration ([EIP], [PofEAA]) have emerged and already been successfully applied to solve real world, large scale integration problems. Modern integration solutions make extensive use of these patterns and enable integration architects and developers to apply them in their own work.

Looking at the *desktop* domain, on the other hand, the situation is substantially different and development towards a similar level of integration is lagging behind significantly, facing problems that have been addressed in

the enterprise almost 10 years ago. This has several reasons: The user interface has much higher priority than in enterprise solutions, which mostly integrate backend systems, and there are no comparable standards for interoperability such as *CORBA* or *SOAP*, and no common, platform-neutral standards for *inter-application messaging* like in the enterprise, e.g. *web services*, or language-specific solutions as in *JEE* (such as *EJB*) or .NET. Desktop applications are inherently bound to the underlying operating system, where approaches like *OLE* and *COM* limit integration possibilities to a single platform and force integration at a low level, where applications directly invoke functions in another application. This results in tightly coupled application "suites" that are static and cannot be changed or recomposed on demand by users. Vendors have traditionally been reluctant to provide open APIs or componentization facilities for their products, as that would make it easy to exchange individual components with products from other vendors or to add missing functionality, reducing the need for upgrades.

As a result, before the proliferation of open source systems and software[1], users have been depending on the goodwill of software vendors to make the software they needed work together in a feasible way. The results were often limited again to mostly *bidirectional* integration among cooperating vendor's applications (e.g., AutoCAD and Cinema4D) that offered a preconfigured and static combination of specific applications in a more hard-wired than "integrated" way. For these and other reasons, which are detailed in Section 2.2, only little advances – mostly in academic areas – have been made in this area, although there is equal demand to integrate applications on the desktop as it is in the enterprise.

In contrast, on the web, the increasingly popular *"mashups"* of the Web 2.0-era [O'Reilly2005] (see also Section 4.3.1) can be seen as innovative examples of *ad hoc* integration solutions, as they integrate different, previously separate web applications to form new "meta-" or *composite applications* that combine the functionality of previously isolated services and data by reusing existing applications, as illustrated in Figure 2.1 below (see Section 4.2 for an example on how these two integration approaches can complement each other).



(from [Hinchcliffe2006])
Figure 2.1: Mashups on the Web and in the Enterprise

Users can freely combine existing applications, such as [GoogleMaps] and [Flickr] to build a geo-tagged photo album. An increasing number of mashup-services like [YahooPipes] or [MicrosoftPopfly] provide visual inter-

[1]even open source software has not yet delivered dynamic integration solutions on the desktop that end users could freely configure, even though KDE4's service-oriented component approach (see Section 3.2.1.3) is promising and worth to be noted.

faces for building *composite services*, whereas *Google Mashup Editor* [GoogleME] takes a more developer-oriented approach. These services (which are listed at popular sites such as [Mashable] or [ProgrammableWeb]) even encourage users to do so by offering pre-built combinations from other users for further customization and by providing a rich, desktop-like user interface based on *Asynchronous Javascript with XML* (AJAX). In this respect, the web now offers a better integrated "desktop" experience by providing user-centric spontaneous integration possibilities which have been created through the proliferation of an open, distributed information architecture (i.e., the Web) using common standards for communication (HTTP), presentation (HTML) and interaction (*JavaScript*), where the browser is the underlying platform.

To summarize, it can be shown that integration challenges on the desktop are quite similar to the enterprise world, where already a wealth of patterns, methods and best practices have evolved over the years, and a variety of frameworks and implementations proven to work in real world scenarios are readily available. An overview is given in Section 2.7 at the end of this chapter. Enterprise integration has since moved away from proprietary, closed and static integration approaches (also referred to as "stovepipe solutions") to service-oriented and highly dynamic approaches that adapt to rapid change as common in the business world. The Web has shown that there is real demand from end users for integrating applications they use frequently, and they will come to expect the same from desktop applications. It is therefore time to look at how suitable and proven patterns and solutions from enterprise integration can be applied to the desktop in a way that is as simple and usable as web based integrated applications. In this work, several of these solutions will be analyzed and it will be shown how the adoption of emerging standards and best practices in enterprise integration can solve the problem of integrating COTS tools on the desktop.

The following section will give a definition of the problem domain, then continues with an evolutionary overview of integration strategies on the desktop, which are described in more detail in Chapter 3. Finally, we will look at how integration is handled in the enterprise, from past to future, and how the solutions developed there could be used to satisfy the special requirements of desktop application integration.

2.2. Defining Tool Integration

"Integrated applications are independent programs that can each run by themselves, yet that function by coordinating with each other in a loosely coupled way.
--Gregor Hohpe, Enterprise Integration Patterns

As integration is a large problem domain, this work concentrates on the aspect of *Tool Integration*, which can be seen as a subset of *desktop application integration* (sometimes also called "system integration" which may be misleading as it does not deal with low-level integration at the operating system level). Tool integration addresses the problem of combining software tools so as to form a dynamic, user-centric workflow, as often needed in engineering. [Thomas1992] provides a general but concise definition of the problem domain, defining *integration* as "property of tool interrelationships", and *tool integration* as follows:

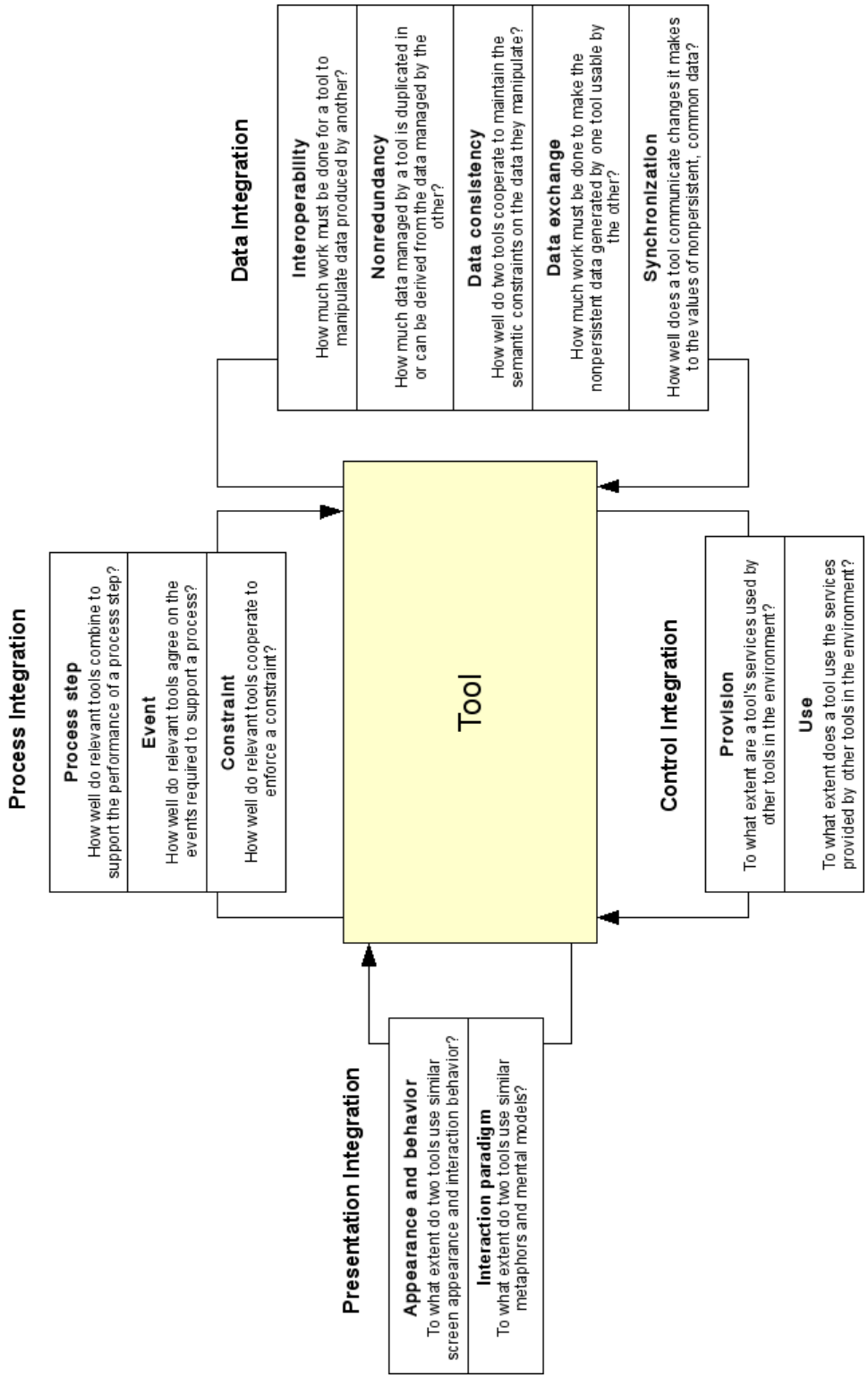
> Tool integration is about the extent to which tools agree. The subject of these agreements may include data format, user-interface conventions, use of common functions, or other aspects of tool construction
>
> —[Thomas1992]

While other definitions mainly focus on the compositional aspect of tool integration, tending towards a single composite software engineering environment, this work investigates how *existing* tools can be integrated in a *loosely-coupled* way, keeping the original tools *as-is*, but linking relevant functionality and data through the original interface by using available tool APIs. *Coupling*, as defined in [Hohpe2006a], is "a measure of the dependency between two communicating entities. The more assumptions the entities make about one another the more tightly coupled they are.". The main principle behind *loose coupling* is to enable high-level collaboration

among applications while keeping dependencies at lower levels at a minimum (c.f. [EIP], p9,39). This results in a stable integration that allows reuse and dynamic recomposition, which enables quick adaptability to new requirements and enhances scalability (i.e., when new tools are added).

The following diagram, Figure 2.2, provides an overview of the terminology and associated integration strategies, which will be introduced in Section 2.3 and discussed further in Chapter 3:

An extensive up-to-date overview of recommended literature, and a good introduction to the problem domain in general, is given in [Wicks2007].



(c.f. [Thomas1992])
Figure 2.2: tool integration dimensions and patterns

2.2.1. Integration of Commercial-off-the-shelf (COTS)-Tools

In Chapter 5, a framework for integrating pre-packaged, mostly closed source *commercial off-the-shelf* (*COTS*) applications, ToolNet, is analyzed and redesigned.

[Goose2000] defines a framework as “a software environment that simplifies the development and management of applications by providing a reusable context for components”, which applies well to the context of tool integration in this work. A general definition of what a *tool* is can be found in [Terzidis2007:147]: “The word *tool* is often used to describe the synergistic interaction of designers with computers. A tool is defined as an instrument used in the performance of an operation.”

Because integration of COTS tools can only happen after the fact, it is also called *a posteriori tool integration*, in contrast to *a priori tool integration*, where tools are designed for interoperability. These two approaches and their merits are discussed in [Barinelli1996] who concludes:

> We argue that, to effectively integrate a tool into tool integration environments, it is necessary to conceive the tool as a collection of services since the very beginning (*a priori tool integration*). *A posteriori tool integration* (e.g., by means of wrappers) could be less effective since a tool is still seen as a monolithic “operator”.

Experiences from research projects and cooperations with the industry have shown that this is an idealized scenario and that a *priori tool* integration cannot be applied on a broader basis. For example, in the engineering domain, existing tools, which are mostly commercial standard tools, often constitute a substantial investment in licenses, training and infrastructure. Such an established tool landscape cannot be easily replaced by custom solutions that may be better integrated but represent new and unproven tools that users are not familiar with. [Altheide2002] also questions the long term use of *a priori integration*, reasoning that “a priori tool integration projects, in the long run, cannot compete with the pace of evolution of commercial stand-alone tools.”. Therefore, tool integration frameworks should allow for an easy integration of existing tools and possibly facilitate a smooth migration path to better integrated tools.

It is important to distinguish general purpose integration frameworks from concrete tool sets or software *suites* which are bound to specific environments or vendors. Such solutions often give the false impression of a modular set of *loosely-coupled* tools, but in reality these are *tightly-coupled* components of a static, monolithic system that provides interoperability only between components of the same application suite, version and vendor. As a result, new tools cannot be added as needed in an easy way, as the data format and tool intercommunication mechanism is often proprietary and complex or even closed. Examples include early browser suites (Netscape/ Mozilla), office suites, software engineering environments that aid development within a predefined software process (Rational Developer, VisualStudio, various SOA solutions), but also cross-vendor suites in the CAD or 3D domain where for example modelers are connected to a predefined set of renderers or other tools in a fixed vendor environment.

This approach is more common in the commercial world, partly because of the lack of suitable component standards (a gap that has been closed in recent years, as shown in Section 3.2.3), but also because of market considerations. By allowing interoperability only with software from one's own company or from partners, vendors have often tried to control “their” market segment. This strategy is often accompanied with strict licensing terms of interface definitions and intellectual property rights enforcement through copyright and patents, a common practice with major market players, reaching from operating systems to APIs and applications (see [Samuelson2006] for a good analysis on this practice and the strategic change in direction with the example of IBM).

Only in recent years, through the increasing adoption of *open source*, the trend is moving towards open systems, APIs and platforms, and modular applications composed out of lean components that tightly focus on a single

task. Examples include the Firefox browser or Thunderbird e-mail client, which have been spun off the Mozilla suite (the open source version of the proprietary Netscape web suite), or OpenOffice that uses an open component specification (UNO, see Section 3.2.3) as the basis for its word processor Writer, the presentation module Impress or the illustration module Draw. Also an increasing number of commercial solutions are being migrated from isolated, monolithic applications to open, plugin-based components or extended with web service interfaces. For example, software development tools like the Rational-suite or even IDEs like Borland JBuilder are moving towards the Eclipse platform, and enterprise solutions from SAP or Oracle offer integration with Web services (e.g., using WSIF, see Section 3.3.3.1).

In specialized markets like engineering however, current tools are still mostly monolithic by design and cannot be refactored to facilitate integration with other tools, because access to the source code is not available or the cost of a custom refactoring would be too high. As a result, several techniques have been developed to provide a service façade for existing tools to the end user and also to tool integrators, wrapping tools into service based *Adapters* and allowing for reuse of tool functionality in other tools.

An overview of current techniques for integrating COTS-tools is given in Chapter 3, whereas the final, service oriented solution is proposed in Chapter 4.

2.2.2. Relation to Enterprise Integration

When viewing tool integration from a more general perspective, it can be shown that there are many similarities to typical enterprise integration problems, e.g., taking the following definition of application integration:

> Application integration […] is the process of bringing data or a function from one application program together with that of another application program. Where these programs already exist, the process is sometimes realized by using middleware, either packaged by a vendor or written on a custom basis.
>
> —from SearchSOA.com[2]

Although this definition is taken from sources related to enterprise integration (see Section 2.6), tool integration on the desktop can be defined in a similar way. One important aspect that is central to this work is emphasized in [EIP]: integrated applications stay independent, but interoperate transparently with other applications, so that their functionality can be extended with functions provided by other integrated applications.

Whereas the related term of *COTS integration* is often used in the context of integrating newly acquired software in an existing system landscape (e.g., [Guerra2003]), avoiding incompatibilities or other side effects, *tool integration* focuses on linking together existing applications on the desktop in new ways and strives to overcome integration barriers imposed by third-party applications such as limited APIs or missing communication interfaces, in a way that enables users to connect their existing toolset in a spontaneous manner, even if the individual tools are unaware of each other and do not interoperate per se.

As outlined in the Section 2.1, current integration approaches (see Chapter 3) are enterprise-centric and focus on integrating custom or proprietary backend systems in a *service-oriented architecture*, which is usually realized through an *Enterprise Service Bus* (*ESB*, see Section 3.3.3.2) or a thin integration layer based on web services[3] or *REST* (see also Section 8.5). In contrast to desktop applications, the target systems usually lack a dedicated user interface[4] and operate transparently in the background, performing business logic involving database access, network transfer and communication with other backend systems. Enterprise Integration on the desktop does not reach beyond specialized solutions like SAP, Microsoft Office or Lotus Notes, which integrate only a limited

[2] http://searchsoa.techtarget.com/sDefinition/0,,sid26_gci211586,00.html

[3] see Section 3.3.7.2 for related standards and Section 3.3.3.1 for a concrete example

[4] although there are interfaces to host-systems, these are mostly terminal based or simple web interfaces where usability requirements are rather low, and integration either targets only a single backend system or provides a simple façade for related backend applications, which is different from the integration requirements outlined later

number of related, vendor-specific backend systems with selected, proprietary desktop applications (usually by the same vendor) in a homogeneous system landscape, leaving a gap or *missing link* to existing legacy applications, which is often termed the *"last mile of SOA"* [OpenSpan].

On the other hand, as has been mentioned before, most desktop applications are not designed for integration with other applications that reach beyond single-vendor software conglomerates or “suites” and thus do not offer extensive public APIs that could be used for integration with other applications. Even when they do, they allow only limited access to the application's functionality, which may be only available to scripts that operate inside the application, and with the limited scope that developers decided at design time. This is often not sufficient to realize an integration solution that meets dynamically changing needs of end users, who require interoperability between applications from different domains in a transparent and usable manner. The problem of integrating isolated applications in a heterogeneous environment is well known wherever several related but disparate applications are used, such as in the aforementioned customer service-domain or in medical institutions, but also in research and engineering, as shown in the examples in Section 3.2.4.

To summarize, this work tries to solve the problem of integrating desktop applications such as *COTS* tools in a direct and spontaneous manner, overcoming legacy issues and API constraints as transparently and effectively as possible, by using a standards based, adaptive architecture that is open to changes in integrated applications or user requirements (see Chapter 6 for a detailed description). An important distinction to enterprise integration is that the original tools should stay autonomous and only be augmented, allowing users to continue using their preferred tool set they are accustomed to, only in a more flexible and interoperable way. The solution presented in this thesis allows existing tools to provide extended functionality to end users by using services offered by other tools. Integrated Tools work together in a transparent way, interconnected through an open and extensible integration bus. The user is freed from manual “integration” tasks such as having to export and import data using common exchange formats supported by related tools in the tool set, compromising individual tools' strengths and limiting the user's choice in selecting tools best suited for the task at hand.

2.3. Terminology: Levels and Patterns of Integration

[Trowbridge2004] provides a detailed analysis of the integration problem (again, looking at the enterprise domain, but easily applicable to desktop integration), dividing integration into corresponding layers and showing how to connect to applications on each layer. For each layer and connection type, the work introduces several *integration patterns* that can be applied to a practical integration problem. Similar classifications can also be found in other sources, such as [Erl2004:288], which covers service-oriented architecture in more detail (see Section 3.3.3). [Amsden2001] adds and *API integration* (which can be seen as a more abstract form of application integration from the classification found in [Trowbridge2004]), which are provided, e.g., by the Eclipse platform (see also Section 3.2.4.2.1).

Table 2.1 below tries to combine the different classifications of integration types found in literature and adds another dimension, the *integration domain*, spanning from the desktop across the network to the enterprise, to illustrate the different situations encountered across integration scenarios with their specific needs and limitations. The vertical axis indicates the level of integration abstraction, which is increasing from the data layer at the bottom to the process layer on the top. With higher levels of integration, the result is usually more effective and provides a better unified user experience, minimizing gaps between tools and eliminating the need for manually performing necessary translation or duplication of information contained within tools. The table also gives examples of current approaches that are relevant for tool integration and detailed in the remainder of this chapter:

Layer \ Domain	**Desktop**	**Network**	**Enterprise**
Process Layer	Automator (MacOS X)	— (no real standard outside of enterprise environments)	*WS-BPEL*, WSCI, ebXML, *WCF*

Layer \ Domain	Desktop	Network	Enterprise
Presentation Layer	OLE (MS Windows), KParts (KDE), Replicants (BeOS), OpenDoc (MacOS)	RDP (MS Windows), X11 (UNIX)	Portals, Dashboards (e.g., management consoles)
Functional Layer	*component (based) integration: COM*/ActiveX (MS Windows); scripting: AppleScript (MacOS)	*Distributed Object Integration:* DCOM (MS Windows), CORBA, RMI	*JCA*, Spring (*JEE*); *OSGi*; *message-oriented middleware integration* (*MOM*); *Service Oriented Integration* (*SOI*):*JBI*, *SCA*, WSIF
Data Layer	*file based integration:* Pipes (UNIX), file exchange/common file formats (CSV, XML), FTP, *ETL*	file transfer (S/FTP, *ETL*)	ODBC/ADO.NET, JDBC; product specific or legacy Adapters; SOI: *SDO* (Service Data Objects)

Table 2.1: Overview of Current Integration Concepts

It is important to note that these layers are not strongly divided but are often combined, e.g., lower levels may be reused to realize higher level integration: As observed in [Gautier1995], boundaries between integration forms are blurred, e.g., process integration may be implemented using control integration, and data integration is almost always involved as it is a prerequisite for higher-level integration. Also, the horizontal axis denoting domains should not be seen as a strict division, as desktop integration concepts become increasingly distributed and even incorporate concepts from enterprise integration, which is also shown in Chapter 3, Figure 3.1. This fact is later exploited in the proposed solution and associated prototype.

The subsequent sections provide a more detailed view on the integration approaches outlined here, complementing the classification with additional concepts and concrete solutions in the field.

2.3.1. No Integration

Sometimes it is not necessary to integrate a specific tool, it may be sufficient to poll for an output or similar. For tool integration this means that the tool is not directly needed as part of a workflow but may produce some artifacts in the background that are retrieved independently by another tool later.

2.3.2. Invocation (Launch) Integration

In this scenario, tools are launched as needed, with parameters being passed during launch. This approach is useful for integrating existing and mostly file-based tools. E.g., for integrating Telelogic DOORS (introduced in Section 5.4), the requirements engineering tool used in the prototype scenario, launch integration is used to start the tool from within the prototype interface on user request. Another example is the Lean Integration Platform (LIP) by Frauenhofer Research[5], which integrates existing tools into a predefined workflow that can be programmed using LISP. Relying solely on launch integration only provides very basic integration possibilities and is not enough for realizing transparent collaboration among disparate tools, as existing functionality and data is still bound to the original tools and not available to other tools in the workflow.

[5] see the LIP product page [http://www.pb.izm.fhg.de/lip/]

2.3.3. Data Integration

This form of integration is used when data has to be shared among several applications that need to operate on the same data. The solution is to “integrate applications at the logical data layer by allowing the data in one application (the source) to be accessed by other applications (the target)” [Trowbridge2004:125].

There are several ways to achieve this data exchange: examples reach from file-based interchange, preferably through open standards such as XMI used in modeling or general XML, to databases (using abstraction layers like ODBC/ADO.NET or JDBC), up to modern high-level data sharing such as the emerging *Service Data Objects* (*SDO*)-standard (see also Section 8.5). This form of integration often provides the lowest common denominator for independent tools to cooperate, and can be implemented with feasible effort. However when used as the sole integration method, it is not the most effective solution as it integrates at a rather low level, resulting in a possible tight coupling among integrated applications.

Although current efforts (see Section 3.2.1.1) allow new ways of working with data and exchanging information between applications, they still face the limitations of *semantic* data integration, namely data source heterogeneity and missing tool support [Gorton2003], the performance penalty inherent to file operations, and concurrency issues when multiple tools want to access the same data simultaneously (compare [Reiss1996]). Also, *a posteriori* tool integration is only possible through filters for importing and exporting files, which requires manual interaction. Lastly, full integration on the user interface-level, which is a requirement for successful tool integration, is not possible. Thus, data integration alone is not sufficient for transparent and efficient tool integration; often, a more feasible solution is *functional integration*.

2.3.4. Functional Integration

This form of integration (also called “control integration” in some older work) operates at the application level, accessing APIs and other interfaces exposed by the target application (c.f. [Trowbridge2004:135]). Unfortunately, as previously mentioned, not all applications provide an API and if they do, the interface exposed is often limited to certain use cases which is impractical for general purpose tool integration. Nevertheless, it is a powerful integration mechanism that is commonly used for tool integration as it provides the most flexibility to integration developers, while at the same time allowing to target a stable, standardized application interface. This results in a reusable and open integration solution instead of a custom low-level integration solution that is fragile and likely to break as the target application changes. An example for a tool integration approach using control integration is given in [Michaels1993].

In recent years, more and more software provides APIs for high-level languages and standard scripting languages[6], as opposed to legacy or C-based interfaces of previous years, and more recently service-oriented interfaces (see Section 3.3.3) that can be accessed as web services.

Component integration (see Section 2.3.4.2 below) is also a form of functional integration and widely used in current architectures such as Java Enterprise Edition (*JEE*) or frameworks like *OSGi*, which is covered in Section 3.2.3.

Message based integration (see Section 3.3.2) is a combination of functional integration and data integration, as applications can exchange information but also send requests to other applications.

As this integration style and combinations thereof are the most commonly used approaches, many examples can be found in desktop and also in enterprise APIs and frameworks, which are covered later in this chapter.

[6] traditional Windows applications tend to use VisualBasic, whereas cross-platform or open source software uses open languages such as Python, Ruby or various Java-like languages (e.g., Groovy or Scala)

2.3.4.1. Application (API) Integration

Traditionally, closed or vendor-specific APIs have been the predominant way to access applications from outside, mostly in the form of binary C/C++ libraries. Alternatively, a scripting interface is often exposed that allows developers to write extensions or macros that run inside the application, but reach outside the application boundary for intercommunication with a backend system or with another application.

On the system level, application-level integration is realized through a mechanism called *inter-application communication* (IAC), which is defined as a “technology that allows different applications in a computer system to effectively exchange data and information, which is the base of realizing software cooperation and software system integration.” [Lan2004]. Realization is often done through messaging or scripting (e.g., AppleScript) and provided either directly by the operating system, or by component frameworks such as ActiveX, UNO, or OSGi (see Section 3.2.3).

A practical example for API integration in an integration framework is given in the prototype scenario (see Chapter 6), where an existing commercial requirements engineering tool is integrated, namely Telelogic DOORS [DOORS] (see Section 5.4). The application exposes access to requirements (stored as objects) through a scripting API and can be accessed from outside using a C library. By wrapping the library functions inside method calls, the C library is made available to high-level languages, e.g., using *JNI* in Java (or the more recent *JNA* library which is much easier to use, see Section 6.4.1.3.3).

2.3.4.2. Component Integration

> *There has been much talk about component architectures but only one true success: Unix pipes. It should be possible to build interactive and distributed applications from piece parts.*
> *--Rob Pike, Bell Labs, Lucent Technologies, 2000*

Component Based Software Engineering, in short CBSE[7], provides a way to break up software into functional units that can be dynamically recomposed as needed. A thorough state-of-the-art analysis of this software development model is given in [Szyperski2002], who defines a component as follows:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
>
> —from *Component Software* [Szyperski2002]

This technique is well suited for tool integration, as it makes available functionality in a reusable, more general form by splitting monolithic software into modules. The provided functionality can then be recomposed in new ways, creating new *composite applications* out of previously isolated and self-contained functionality. Modern *component-frameworks* (further covered in Section 3.2.3) make available this composite software paradigm to application developers and integrators. A definition is given below:

> A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be "plugged" into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances.
>
> —from *Component Software* [Szyperski2002]

Hence the term *plugin frameworks*, which emphasizes on the dynamic aspect of component based integration frameworks, which allow recomposition of components at runtime. Dynamic composition is a key aspect related to late binding in programming, where dependencies are resolved at runtime, not at compile time. This makes

[7] also called CBSD, for component based software *development*

component integration a viable choice for *a posteriori* tool integration, where existing tools or "components" cannot be changed or adjusted but have to be integrated as is at runtime.

2.3.5. Presentation Integration

Whereas data integration (see Section 2.3.3) operates on the application level, invisible to users, this form of integration is most visible to users, as it integrates applications at the user interface level. For this reason it is also called (user) interface or UI integration in literature (e.g., [Brown1992], [Amsden2001]). This technique was used in the past to integrate host systems that offer a terminal interface (e.g., IBM 3270 systems, also called *green screen* systems), importing data by copying and parsing text from host screens – hence the term *screen scraping* – and exporting data by simulating input over the terminal. Presentation integration is not limited to legacy integration, but is also used in GUI testing, e.g., JMeter[8], a web application test tool, or the Linux Desktop Testing Project[9]), which uses existing accessibility libraries to control applications via the user interface. Also portal integration (dashboards) can be seen a form of presentation integration.

2.3.6. Process Integration

Process integration "provides orchestration of activities across multiple applications according to predefined business processes […]" [Trowbridge2004] but in the context of tool integration can be defined more generally as the integration of the functional flow of processing between applications. Because a major motivation for tool integration is to facilitate a seamless workflow for users of individual tools, this form of integration blends naturally with many goals in tool integration. In the enterprise domain, recent XML-based standards like *[WS-BPEL]* and WS-Orchestration, and to a lesser extent *Wf-XML* and *XPDL*, facilitate process integration in an SOA, as described in more detail in Section 3.3.4. Also the Spring framework (see Section 3.2.3.2) provides several examples of process integration, as it "wires" together JEE components in process-oriented ways, e.g., Spring WebFlow[10]for web applications, Spring Batch[11]for batch processing, or Spring dm for OSGi-based applications.

2.3.7. Model-Driven Integration

Currently, the highest level of integration works at the model level and is a form of *model-driven engineering* (also called *model-driven development*), as introduced, e.g., in [Voelter2006]. [Mellor2003] defines a model in a general way as "a coherent set of formal elements describing something […] built for some purpose that is amenable to a particular form of analysis […]". Consequently, model-driven development is defined as "the notion that systems can be developed by constructing abstract views of systems and by transforming the resulting models, either automatically or manually, into code."[12]. Model driven (or model based) integration applies this development model to solve integration problems in a more general way, abstracting from specific implementation details:

> Model-driven integration differs from the programmed integration. Programmed integration relies upon hard-coding a finite, and inextensible, solution to a particular challenge. Model-driven integration focuses on abstracting the information content into a model that describes the enterprise's information resources. This model captures the nature of the information the enterprise has within its systems and the way the enterprise uses data in its daily operations.
>
> —from the article *Model Driven Information Architecture* by Brian J. Noggle and Michael Lang, available at TDAN.com[13]

[8]see Apache JMeter [http://jakarta.apache.org/jmeter/]
[9]see Linux Desktop Testing Project Wiki [http://ldtp.freedesktop.org/wiki/]
[10]see the Spring WebFlow home page [http://www.springframework.org/webflow]
[11]see The SpringBatch home page [http://static.springframework.org/spring-batch/]
[12]from the call for papers for Model-Driven Development, Special Issue Publication: September/October 2003
[13] http://www.tdan.com/view-articles/4989

A major advantage of using models is seen in reusability of expert knowledge at a high level: [Mellor2003] argues that “modeling is an appropriate formalism to formalize knowledge” and that “model-driven development captures expert knowledge as mapping functions that transform between one model and another.”, thus decoupling domain knowledge from the concrete implementation and allowing reuse across different platforms and implementations by reapplying the model and related mapping. This view is backed by [Schmidt2006], who provides a short analysis on why earlier *Computer Aided Software Engineering* (CASE)-efforts failed, observing that common languages and platforms “provided abstractions of the solution space […] rather than abstractions of the problem space”. *Model transformation* is applied in many tool integration solutions for mapping between different tool's data models and for keeping models synchronized when changes occur. [Tratt2005] provides an introduction to model transformation and available solutions, such as the *OMG* standard QVT (short for Queries, Views and Transformations), which defines languages for *model-to-model* transformations.

The advantages of model transformation are also emphasized in [Kramler2006] who concludes that “model transformation techniques […] avoid the pitfalls of strongly technology-dependent solutions that suffer from high maintenance overheads and most importantly poor scalability.”. Current solutions built with imperative, general-purpose frameworks and APIs are seen as too complex and error-prone, because they still largely follow an *imperative* paradigm and require much handcrafted “glue” code and configuration. This results in a *fragmented* view that “forces developers to implement suboptimal solutions that unnecessarily duplicate code, violate key architectural principles, and duplicate system evolution and quality assurance.”. Model-driven engineering could provide an *integrated* view that closes the *semantic gap* between design intent and implementation, and related integration issues (deployment, configuration and testing), following a *declarative* paradigm based on *domain specific modeling languages* (DSMLs) that are more suited for expressing domain concepts than general-purpose languages.

Models can again be specified using meta-models (e.g. UML itself is specified by the UML Metamodel), using the *OMG* Meta-Object Facility (MOF)-standard and QVT. The MOF allows extending and adapting models (or modeling languages itself, like UML) to different domains and usage profiles. E.g., [UMLEAI] defines a UML profile for *enterprise application integration* (*EAI*), suitable for modeling SOA solutions (see Section 3.3.3). These specifications are part of a general modeling standard, the model driven architecture (MDA), as specified by the OMG[14].

[Schmidt2006] provides a review of the current state-of-the-art in model-driven engineering, including two case studies that show practical examples of how the model-driven approach can be applied in complex real-world integration scenarios. Other examples for real-world solutions based on model-driven concepts will be covered in Section 3.3.6, together with a critical evaluation of model-driven integration.

2.4. Examples of Tool Integration

Probably the most prominent example for an integrated desktop application is an *Integrated Development Environment* (IDE): Previously, software developers had to resort to individual tools for each development activity, ranging from source code editors to compilers, linkers and debuggers. Each part of the *tool chain* had to be individually configured, invoked and mastered for every software project, and output from one tool had to be manually transferred to the next tool in the chain. Software developers had to manually interpret errors in the process from output on the console, and look up the matching location in the corresponding source file. Early integration efforts provided ways to invoke tools from within a source code editor and jump to the corresponding error location when available, but only recent solutions like Eclipse, IntelliJ IDEA or VisualStudio show the possibilities and advantages that fully integrated tools (integrating at several levels including the user interface) can provide to desktop users.

[14] for more information, see the MDA Guide working page [http://ormsc.omg.org/mda_guide_working_page.htm]

More conventional examples include integrated software *suites*, where applications that need to work together are pre-assembled by software vendors to form an “integrated” application, such as an office suite that usually combines a word processor, a spreadsheet application and a presentation program. Scripting or macro languages and APIs provided by the suite's components are the only way for users to realize dynamic custom integration needs despite the static configuration of the pre-assembled software composite.

It is important to note however that – except for Eclipse, to some degree (using a component-model based on OSGi, see Section 3.2.3.1), and truly modular, *component based* applications like KOffice that make their components available to other applications – even modern, “integrated” applications only offer limited ways of interacting with other applications, mostly through *scripting* facilities that are often bound to a single vendor solution or platform. This is not enough for enabling users to freely combine the functionality of disparate desktop applications in heterogeneous system landscapes, so as to dynamically form integrated and task-oriented workflows according to current project needs. Also in a distributed environment, teams should be supported in collaborating using a custom tool chain, which reaches into the discipline of *computer-supported cooperative work* (CSCW).

As a concrete, simple example for a *desktop integration* problem in the sense of this thesis, consider the following scenario: The author uses a tool for writing this thesis in *DocBook* format[15], and another tool for managing references[16]. Although both tools are realized in the same language (Java) and run on the same platform, they cannot be easily integrated, e.g., to allow for lookup and automatic insertion or auto-completion of references, and the references-manager does not indicate if a reference is used in the thesis document, or how often, and in which location[17]. It would be highly desirable to integrate these tools for improving and automating the author's workflow. Although both tools are Java-based, direct integration, e.g. via scripting or RMI (Java's *Remote Method Invocation* communication standard), is not desirable as it leads to a tightly coupled solution that quickly degrades into a unmaintainable *point-to-point integration* that does not scale as more applications are added.

This problem was also encountered in *enterprise integration* (see Section 2.6.1 below) and led to the introduction of *middleware*, acting as a *mediation* layer and providing a common bus for communication between applications. A common message bus decouples applications from static point-to-point connections, because they are connected to a shared bus instead of directly interfacing with each other. As a conclusion, it can be seen that there is equal need for high-level integration in the enterprise as there is on a smaller scale, on the end user's desktop, and that there is a lot to gain from the lessons learnt in the former when applied in a suitable manner to the latter.

Another more complex, real world example is *application integration* in call centers, where agents usually have to deal with a mixture of various heterogeneous applications, partly web-based and partly client-based, maybe even mainframe-based, interfaced with terminal emulators. Such a desktop-mix has several disadvantages which reduce productivity and raise the cost of development, maintenance and use: As older applications are mixed with newer applications, deployments become unstable, resulting in performance hits for both groups and conflicting operating system-dependencies (version-mismatch in libraries, or need for compatibility-layers and other workarounds that generally degrade stability and performance). The user is faced with different interface-paradigms (from console-based mainframe systems to recent web-based, AJAX-style applications, often using isolated clip boards etc.), the need for repeated logins, duplicate data entry and lookup, and other (e.g., semantic) discrepancies.

A final example are hospital systems, which are mostly run on mainframes and interfaced with terminal emulators. This is a poor solution for end users, who have to deal with untypical response times and archaic user interfaces[18].

In Section 3.2.4, currently available solutions will be investigated further, including a solution to the last example.

[15] using a free version of XMLMind's excellent XMLEditor [http://www.xmlmind.com/xmleditor/]
[16] the free BibTeX tool JabRef [http://jabref.sourceforge.net/]
[17] With version 2.4, there is now an OpenOffice plugin that aids users in citing references stored in JabRef, and JabRef provides a command line option for inspecting LaTeX's .aux-files for references, building a list of references that are used in the document.
[18] The author has had the pleasure to work with such a system ("KISS") in a local hospital during civil service.

2.5. History of Tool Integration

[…] the user's workflow becomes automated. Instead of performing one task with one application and then inputting the results into another application to perform the next task, users focus on solving a whole problem, not performing a series of tasks that result in the problem's solution.
--ToolTalk Whitepaper, Sun Microsystems, Inc.

Even long before the notion of a *desktop* even existed, the UNIX™ operating system used a technique called *pipes* to connect independent command line programs, directing the output of one program to another, which interpreted it as input. Several programs could be daisy-chained and complex sequential actions could be realized by using simple operators. This is one of the simplest and most effective integration solutions still used today, which has been described as the “Pipes and Filters”-pattern in [GoF]. In terms of integration, the technique is a basic form of file-based integration, as the operating system uses special files for realizing input and output channels that connect individual programs to form a tool chain, much like composite applications are connected in today's *service-oriented* architectures[19].

For this pattern to work, software must be written with a *tool*-based approach, where each individual software application solves a certain problem but offers generic interfaces to allow for composition with other applications, facilitating a *divide-and-conquer* approach, where a complex problem is recursively divided into subproblems until the resulting problems are trivial to solve. The idea or pattern of software tools was proposed as early as 1976 in [Kernighan1976], who provides various examples on how to design and implement software as tools.

However, terminal applications that used "interactive" text-based screens controlled by user input could not be integrated with this approach, and only raw streams could be exchanged, which required knowledge of the data format from both the source and target applications. This created a *tight coupling* that became problematic when new data formats were introduced. As long as small and specialized command-line programs (like tar, gzip and more) were used, this was an acceptable compromise, as new formats could be handled by inserting another command into the pipe that acted as a translator. With increasingly complex applications and the commercialization of the software landscape, this solution no longer worked and users became increasingly involved in ensuring that programs worked together as needed.

The advent of graphical user interfaces and the introduction of new operating systems and programming languages brought yet more challenges in integration, as applications could no longer assume a common data format or communication mechanisms. They also became independent of a command line acting as a single point of control. Various forms of user input were employed and applications exposed only a subset of the internal functionality in order to simplify user interaction, which limited possibilities for *inter-application communication* (IAC) and shifted responsibility to the user who now had to take care of transferring data between applications.

In the early 1990s, commercial companies like Sun identified the problem of integrating applications on the (UNIX) desktop and the need for a transfer of control to users. They realized a distributed, object-oriented, message-based API to enable inter-application communication [ToolTalk], that was later incorporated into the *Common Open Software Environment* (COSE), a multi-vendor initiative to provide an integrated desktop API for UNIX:

> The Common Open Software Environment (COSE) Desktop offers several API's and tools to allow application programmers to integrate their programs with Desktop services. The invention disclosed ties all of these various Desktop tools and services together to provide one place in a Desktop Application Builder tool where the developer can "step through" the process required for Desktop application integration.

[19] this analogy is also supported by SOA expert Tin Man in his blog article SOA and UNIX [http://blogs.sun.com/tientien/entry/the_soa_philosophy_a_new]

—from the original press release, retrieved from The Prior Art Database[20]

The initiative eventually resulted in the *Common Desktop Environment* (CDE) which was the predominant UNIX desktop until open source alternatives such as KDE[21] (which is in fact a wordplay on CDE) and later GNOME[22] emerged in the late 1990s.

The introduction of networks resulted in the creation of new integration forms at a higher level, abstracting from concrete programming languages or local communication mechanisms. RPC implementations like [CORBA] (which was based on the remote object invocation protocol in ToolTalk) have later emerged to solve the issues of distributed inter-application communication, but the problems of accessing graphical and packaged applications remained.

Around the same time, the *FIELD Environment* [Reiss1990] tried to solve the problems of integrating desktop applications in the software engineering domain and proposed a precursor of modern integrated development environments: “FIELD demonstrates a simple but effective way to unite many existing tools in an integrated programming environment” (ibid.). The framework connected several custom development tools through TCP/IP sockets, allowing for distributed collaborative work, and provided a simple but efficient solution to inter-application communication by using *message-passing*, an integration-technique which is now called *message-based integration*. This concept was rather novel at that time and is still used today as part of higher-level integration solutions. Applications could register for messages of interest (using pattern-matching) and were informed by a central *message router* using a mechanism called *selective broadcast*. A graphical frontend allowed interactive operation of the integrated development environment, and new tools could be integrated by adding suitable functions for handling FIELD-messages.

While the FIELD-framework provided an open and extensible approach to *control integration* by integrating tools through messaging, it was necessary to modify the source code so that existing tools could be extended, for making them available to the framework and other integrated tools, which is not an option for integrating existing, closed tools. Also, *data integration* was only rudimentary implemented, as tools had to convert to and from messages from other tools by themselves; there was no abstract message format that could be handled in a uniform way (as in modern integration frameworks like JBI which use *normalized messages*, see Section 4.2.1). Data was only treated as a set of parameters for control messages, but not handled at a more semantic level, which would allow for connecting related data elements and for providing a storage facility (or *repository*) for common information shared between integrated tools.

While integrated development environments[23] were the primary focus of early desktop integration efforts, also *hypermedia* environments faced similar problems, as they had to enable knowledge workers to work with different tools in a seamless and transparent way. The goal was to build a coherent information space of the target domain, where different artifacts could be connected with each other in a meaningful way: HyperDisco [Wiil1995] approached this challenge by offering different levels of integration, which would “allow different tools to be integrated in the hypermedia framework at different tool-dependent levels. Instead of providing a single model of integration that all tools must adhere to, we allow each tool to have its own specialized model of integration and its own specialized protocol for accessing the hypermedia services.”

Another related hypermedia framework, MicrocosmNG [Goose2000], identified early that “it's becoming increasingly common for information workers to interact with a variety of applications hosted on a diverse range of computing platforms” and proposed *links* as a possible solution, concluding that “such systems must support cross-platform linking through heterogeneous application integration.” As an example, the framework was used to integrate Microsoft Word on Windows (using a VisualBasic macro that adds a menu to the Word interface)

[20] http://www.priorartdatabase.com/IPCOM/000113611/
[21] see The KDE home page [http://www.kde.org/]
[22] see The GNOME home page [http://www.gnome.org/]
[23] also called *Software Development Environments* (SDEs) in earlier work

with Emacs on UNIX using Lisp (also adding a menu). These custom added application services then extract the current selection and transmit the content to a Link Service, which correlates the common information managed independently by the integrated tools. The work correctly identifies integration of existing tools as one of the remaining challenges, which represents the primary obstacle in tool integration in general: with pre-packaged software or legacy applications, it is not easy to add functions as needed, and without access to the source code, alternate ways have to be found for circumventing limitations in functionality and in the API (if available at all), so that integration of such applications becomes possible and feasible. The concept of linking together related or corresponding information divided by tool boundaries is a major factor in successful *tool integration* and has been widely used also in other integration frameworks and domains (such as the ToolNet-framework introduced in Chapter 5).

More recent approaches to desktop and tool integration will be described in Chapter 3, including solutions from enterprise integration, which form a major contribution to the tool integration space in general, as explained below.

2.6. A Short Introduction to Enterprise Integration

> Application integration (sometimes called enterprise application integration or EAI) is the process of bringing data or a function from one application program together with that of another application program. Where these programs already exist, the process is sometimes realized by using middleware, either packaged by a vendor or written on a custom basis. An common challenge for an enterprise is to integrate an existing (or legacy) program with a new program or with a Web service program of another company.
>
> —taken from SearchWebServices[24]

Before concentrating on the core problem of this work, *desktop application* or *tool integration*, it is important to understand where most of the currently available integration concepts, patterns and best practices originated: the *enterprise domain* was and still is a big driving force behind integration efforts for solving complex integration problems encountered in the industry. After huge investments in large-scale integration projects, many of which failed or did not provide a durable solution, working best practices and standards evolved, like *Service Oriented Architectures* and *web services*, open interoperability-protocols like SOAP, and open data interchange formats like XML.

Examples of enterprise integration spawn across all domains, especially the financial sector where banking and insurance systems have to be integrated, often as a result of two companies merging or entering a partnership, but recently also the mobile sector faced complex integration problems when many existing legacy systems had to be migrated to 3G networks. A challenge in this sector is the combination of several previously disconnected mobile services to enable value-added services like location-awareness or interactivity.

Solutions from the financial sector, especially the insurance domain, include the Enterprise Service Bus (*ESB*), which is described below, and concepts like service-oriented integration (*SOI*). In the mobile enterprise, standard efforts like JAIN SLEE[25], strategic alliances like the OMA and solutions like OpenCloud have emerged.

While these concepts and solutions cannot be blindly applied to the problems of desktop application integration (see Section 3.2), a lot can be learnt from the cumulative integration experience gained in the enterprise, and existing standards and solutions can be reused (see Chapter 4).

[24] http://searchwebservices.techtarget.com/

[25] see the article JAIN SLEE Principles [http://java.sun.com/products/jain/article_slee_principles.htmlhttp://java.sun.com/products/jain/article_slee_principles.html]

2.6.1. The Past: The EAI Legacy

In the enterprise domain, the need for integration became apparent with, among others, the Y2K problem, when a lot of big and complex legacy systems had to be analyzed and adapted to function correctly when the year 2000 arrived. Also mergers and partnerships often caused high costs and much more effort than anticipated, because incompatible systems had to be replaced or adapted to make them work together. In many cases, the bigger picture of integration was yet unknown, and the organizational structure was not ready to overcome existing boundaries and embrace the model of dynamic *business processes* that span across departments and management levels. The result was a heterogeneous landscape of isolated applications with proprietary protocols and interfaces. Often, the inner workings of these legacy systems were neither known nor extensively documented, and the people that implemented them had long since left the company. Also, time constraints pushed for "quick and dirty" ad hoc solutions which often resulted in hard-coded bridges that directly integrated one legacy application with another. Over time, as more and more systems had to be integrated, this approach led to an unmaintainable, inefficient and costly conglomerate of *tightly coupled* applications, a method which is now known as the *anti-pattern*[26] of *point-to-point integration* [Sutherland2002].

Even where time and money was not so constrained (e.g., large financial institutions or insurance companies), integration methods and patterns were yet largely unknown, and big integration projects were undertaken that resulted in expensive *integration silos* or *stovepipes*, aggregating legacy systems and putting them under control of a "universal" *broker* that interfaced with all applications to be integrated. The complexity of interfacing with an increasing number of legacy applications was thus only moved to another area, but not solved.

In absence of suitable standards, methods and best practices in integration, most early integration projects that paid attention to design developed *accidental architectures* (see [ESB:28]), like "hub and spoke" [ESB:118-119], which used mostly proprietary protocols and interfaces, thereby creating new legacy systems that would have to be integrated again in a few years. One common approach was to use *message oriented middleware* (MOM, see Section 3.3 for a current definition), an approach that facilitates *data integration* in heterogeneous environments and connects existing applications using a common message format (c.f. [ESB:77]); e.g., transaction systems like CICS (IBM) or Tuxedo (BEA) use a common, binary format to exchange information, based on strict definitions of data boundaries. These formats were mostly proprietary and vendor-specific, which resulted in closely coupled systems that were tailored to a specific environment.

On a more general level, common specification standards like ASN.1[27] provided a meta language that has been used for defining common, industry-wide standard formats like SNMP[28], the *Simple Network Management Protocol*. Now, these formats are only used in specific domains and in performance-critical situations, or in environments where legacy applications have to be accessed, and the new universal meta-language has become XML, the *extended markup language* specified and maintained by the W3C. [ESB:60-76] provides a thorough analysis on why XML has become the foundation of modern integration solutions, with the major advantages of being human-readable, extensible without breaking interfaces, and facilitating a standardized data exchange among disparate systems.

More recent integration efforts like [Maheshwari2003] tried to solve the problem of integrating legacy applications by using *CORBA* for protocol abstraction and the then rising XML-standard for data-abstraction. The problem with approaches like this and CORBA in particular is that while the protocol is abstracted through a standardized interoperability-layer (IIOP) , integration still happens at the low level, because there is a direct mapping between function calls. This creates a tight coupling to the applications' internal architecture and thus leads to a fragile solution that is not open to change, e.g., when two applications are merged and the internal ar-

[26] defined by Jim Coplien as "something that looks like a good idea, but which backfires badly when applied." (from the c2 AntiPattern-wiki)
[27] see ASN.1 Organization [http://www.asn1.org/]
[28] as SNMP allows distributed management of various devices and networking applications through standardized messages, bridging different OS platforms and applications, it is itself an early example for message-based integration

chitecture is refactored. Also, CORBA mostly mandates *synchronous* method calls, forcing the caller to wait for the method invoked to finish and return control, which is often undesirable in a distributed environment or when it is unclear how long the method call takes to complete. [Henning2006] provides a thorough discussion on the inherent problems with using CORBA for integration, which caused the once popular middleware component standard to become largely obsolete today. The Section 2.6.2 shows how these limitations have been overcome with the introduction of *service-oriented architectures* and related concepts and frameworks.

2.6.2. The Present: Service Oriented Architecture and the Enterprise Service Bus

"By 2008, SOA will be a prevailing software engineering practice, ending the 40-year domination of monolithic software architecture."
--Gartner Group

The success of XML during the late 1990s and its wide adoption as an open interoperability standard for data exchange led to the development of higher-level standard based on XML. A milestone for open, standards-based integration in the enterprise was the introduction of an XML-based interoperability-protocol that led to a paradigm-shift in enterprise integration from *functional* to *service-oriented* integration: applications were no longer seen as a collection of objects and functions, but as services that interfaced at a higher level. These *Web Services* could be described in an independent form (specified by the *WSDL*-specification) and once they had been published to a central repository (which was described by the *UDDI*-specification), they could be accessed using a standard protocol, *SOAP*, and combined as needed by current business requirements. Business processes were no longer seen as static rules limited to organizational entities, but as agile processes that were dynamically changing. This new way of thinking led to the creation of a new discipline known as business process modeling (*BPM*)[29], which is now implemented by standards like the XML-based *Business Process Execution Language* (*BPEL*), as described in Section 3.3.4.

Together, these efforts have changed the way integration is done in the enterprise, and on a larger scale they caused a paradigm shift from an object-oriented architecture and distributed objects of the 1990s era to a *Service Oriented Architecture* (*SOA*), which is explained in more detail in Section 3.3.

On the implementation-side, this new architecture was supported by the advent of a high-level integration backend, the *Enterprise Service Bus* (*ESB*), which acts as the spine of an SOA. Although existing point-to-point architectures could be service-enabled with web services, as described in [Erl2004:326], this is not enough when integration is needed at a higher level and complex interaction between several applications is needed. By connecting legacy applications to a message-based service infrastructure using recent integration standards like JCA, JMS, or web service-Adapters, communication can be transparently handled through a common, extensible and open integration backbone, as described in [ESB:212].

During past integration projects, where a variety of different system architectures and interaction styles had to be integrated, a set of patterns and best practices evolved that had proven successful and were suitable for reuse in other integration scenarios. These patterns and strategies have been collected and described in [PofEAA], which offers a rich collection of design patterns that can be rapidly applied to a wide array of integration challenges, and also in [EIP], which focuses on asynchronous messaging patterns, but also acts as a thorough introduction to enterprise integration in general. Section 2.6 describes several of these patterns in more detail and applies them to solve the integration problems identified here in the proposed solution later.

[29] sometimes also called *Enterprise Business Modeling*, see also EnterpriseUnifiedProcess.com [http://www.enterpriseunifiedprocess.com/essays/enterpriseBusinessModeling.html], which extends the *Rational Unified Process* (RUP) with seven disciplines targeted at modeling problems in the enterprise domain

2.6.3. The Future: Integration Frameworks and Event Driven Architecture

As will be shown later in Chapter 4, existing standards, patterns and solutions alone are not enough to solve complex integration problems, as they still require manual combination and “glue” code to form the desired integration solution, which is now often based on a service-oriented architecture. Also, not all aspects of integration are covered by current standards, such as lifecycle management, reliability, access control or remote administration. In the service-oriented world, the second generation web-service standards WS-* (for *WS-ReliableMessaging*, *WS-Security*, *WS-BPEL*, and other related efforts) try to address some of these problems by extending existing SOA technologies. These developments, which are often summarized under the architectural term *service oriented enterprise* (SOE), are coordinated by the *WS-I* organization that defines profiles and testing tools in order to ensure ongoing compatibility between different web service-implementations, in order to prevent a fragmentation of the standard, as happened in the CORBA world (c.f. [EIP:4]).

Nonetheless, interoperability between web services alone will not be enough to solve integration at a general level, which requires fundamental changes in architecture. Evolutionary approaches like SOI (Section 3.3.3) apply existing SOA techniques to integration, whereas developments like the *event-driven architecture* (*EDA*, see Section 3.3.5) represent a more radical approach that redefines the principles of communication in composite distributed systems and inverts the flow of control by using event listeners instead of direct service or method calls.

In the .NET-world, the ESB architecture is extended to the internet and transformed into an *Internet Service Bus* (ISB), which connects geographically separated businesses through secure channels and allows distributed *orchestration* of services [BizTalk2007]. Frameworks specifically targeted at service composition and integration begin to appear, such as the *Windows Communication Foundation* (*WCF*) for the Windows/.Net-platform, and more generic, cross-platform solutions like the *Service Component Architecture* (SCA) and *Java Business Integration* (JBI), a Java-solution that strives to solve integration at a high level but with using existing standards and concepts, fully embracing successful enterprise integration patterns and solutions and providing them to integration developers in form of generalized but readily applicable APIs and tools.

Some of these technologies are already applied in current solutions, which are described in the following chapter, while the latter mentioned integration frameworks are not yet widely adopted, and there is only little experience available on applying them to real world situations. Two current examples mentioned before, SCA and JBI, will be introduced in Section 3.3.7, whereas the proposed tool integration solution which is based on JBI will be examined in Chapter 4. Other promising solutions that were not yet ready for adoption in the prototype are covered in Chapter 8.

2.7. Desktop vs. Enterprise Integration

As can be seen from the previous sections, both the desktop as well as the enterprise domain have developed special design patterns and technical solutions for individual problems, and there is a big overlap in terms of application integration: Adapters like JCA, component frameworks, e.g., OSGi[30] or Spring, and composite application-frameworks like SCA or JBI can be used for integrating applications in the enterprise but also on the desktop. One important difference is that enterprise integration focuses more on data integration and composition of backend systems or services, whereas integration on the desktop is about improving the user's workflow and productivity. This results in different integration requirements on the desktop, e.g., usability and transparent integration, which are not a major concern when accessing enterprise backend systems. For *tool integration*, it

[30] which plays a special role since it originates from the embedded systems domain, then moved to the desktop (with Eclipse as the main driving force) and now reaches into the enterprise, see also Section 3.2.3.1

is often desired to keep the original tools as is, but extending them in order to better cooperate with other tools so as to align more smoothly to the user's design process or task at hand.

Both domains aim at maximizing *return of investment* (ROI) by reusing existing applications and adapting them to new and changing requirements with minimal effort. Replacing existing applications is usually not an option, as this would require new acquisition of products, resulting in costly analysis, deployment, downtime, and retraining of end users. Modification of legacy applications is also usually not an option, because the source code or the skill set is not available, and even in the case of an open source or in-house solution, it takes developer resources and time to develop a suitable integration path. Also, as noted before, organizational constraints often lead to “quick and dirty” solutions that are fragile to change and have to be replaced when one of the integrated applications changes.

The comparative overview in Table 2.2[31] illustrates the differences and similarities between application integration on the desktop and in the enterprise (for legacy and modern SOA environments):

Req's\Domain	**Enterprise (Legacy)**	**Enterprise (SOA)**	**Desktop Systems**
Platforms	CORBA, CICS/IMS, Tuxedo, TIBCO	.NET, JEE	.NET, Java SE, RCP, proprietary
Programming Languages	COBOL, C++	C#, Java	C#, Java, C/C++, scripting languages (VisualBasic, Ruby, Python, Java-like languages)
Architecture	monolithic/centralized	distributed	monolithic (changing)
Transport	IIOP, MQ, TibRV, JMS, TUX, others	HTTP, RMI/IIOP[a]	OS specific, proprietary protocols (COM/COM+, `UNIX` sockets/pipes)
Payload	binary (fixed, IIOP, TibMsg, FML)	XML (SOAP)	binary (proprietary, OS-specific formats)
Inter-Application Messaging	CICS/Tuxedo Services, shared library calls	Web Service calls, REST; SCA, JBI	IPC (using TCP/IP or sockets/pipes), shared library calls
Communication	mostly synchronous	mostly asynchronous	mostly synchronous
Latency / Short Response Time	low / low priority (batch updates)	medium / medium priority (asynchronous messaging)	low / high priority (user interface)
Security	homegrown, LDAP, RACF[b] etc.	WS-Security, Kerberos, JAAS[c], etc.	varies between operating systems from none to complex multiuser-settings with ACLs, memory and process protection and user rights restrictions
System Management	BMC Patrol, Tivoli, CA Unicenter, NAGIOS, HP OpenView; SNMP	WSDM[d] and Web Services Management Tools (e.g. CentraSite), JMX/JEE application server consoles	environment specific: JMX for Java, WMI on Windows/.NET, proprietary tools for system process control, managed system services

[31] originally taken from a presentation by Jody Hunt at IONA, “Extensible Integration for Software Providers to the Mission Critical Enterprise”, table "The Extensibility Gap"

Req's\Domain	Enterprise (Legacy)	Enterprise (SOA)	Desktop Systems
Session Management	stateful	stateless	stateful
Transaction Management	ACID transactions	“fire and forget”	— (no common concept of transactions; proprietary, application-specific solutions)
Resiliency	Load balancing, failover, disaster recovery	— (vendor/product-specific, many ESBs support at least load balancing through clustering)	—
Standards Support	proprietary vendor or “de facto” standards; *vendor driven*	built upon standards set by international organizations (W3C, OASIS); *community driven*	OS-dependent: closed, proprietary formats, only some common file formats; rapidly changing through open source adoption; *vendor driven* (closed source) / *community driven* (open source)

[a] see Java RMI over IIOP [http://java.sun.com/products/rmi-iiop/]

[b] IBM's Resource Access Control Facility [http://www-03.ibm.com/servers/eserver/zseries/zos/racf/]

[c] Java Authentication and Authorization Service [http://java.sun.com/javase/6/docs/technotes/guides/security/]

[d] an OASIS standard, stands for Web Services Distributed Management [http://www.oasis-open.org/committees/wsdm/]

Table 2.2: Comparing Integration Requirements in the Enterprise and on the Desktop

2.8. Summary

This chapter has given an introduction to the manifold issues of integration, spanning from enterprise to desktop integration, and investigating what can be learnt from the former to design successful solutions for the rather new topic of desktop integration, the main topic examined in this thesis. A short overview of previous desktop integration approaches has been provided, hinting at current solutions in the field, which are detailed in the next chapter. Finally, an overview of enterprise integration has shown what problems have already been addressed and, to some extent, solved in the enterprise. A look at the state-of-the-art in enterprise integration was followed by a sneak peak into future trends and developments, which will be addressed later in this work. With this background information, current developments described in the next chapter can be grasped more easily and the rationale for the final integration solution investigated in the prototype-design can be seen from a broader perspective.

Chapter 3. Current State of Integration

"It is impossible to implement one tool that supports all activities in software development. Thus, it is important to focus on integration of different tools, ideally giving developers the possibility to freely combine individual tools."
--from [Damm2000]

This chapter provides a survey of current approaches to integration, reaching from isolated, proprietary and platform-specific desktop solutions to open and cross-platform standards based on a service oriented architecture (*SOA*). Current and emerging solutions from enterprise integration are examined and evaluated for applicability to the problems of tool integration as outlined in the previous chapter. Concrete implementations, such as [ToolNet] (which is described in more detail in Chapter 5), as well as academic and commercial offerings are presented as case studies for the state of the art in desktop and enterprise integration.

3.1. Introduction

Building on the analysis in Section 2.3, which divided integration concepts into *horizontal* layers, focusing on a more general definition of the problem domain, this section arranges current approaches in *vertical* layers, emphasizing on concrete forms and implementations of current integration solutions, combining several strategies and crossing different domains, reaching from the desktop (which has been the focus of most tool integration frameworks so far) to the web and the enterprise domain.

While on the desktop, only few standards applicable to (tool) integration have been developed, and hence most of the solutions outlined in the section below are proprietary or platform-specific, both the *enterprise* domain and the *web* have developed rich standards for data interchange and open communication patterns, reaching beyond traditional databases or shared repositories, by using open protocols such as SOAP or HTTP, and data formats such as XML.

Figure 3.1 shows the increasing overlap of the three major domains relevant to integration and what solutions are currently available. This view is also shared by [Brown1992], who concludes that "none of these levels alone captures the complete notion of integration" and that these levels can be viewed as "independent dimensions within an integration model", hence the need for a *holistic* view of integration. The standards and implementations mentioned will be explained further in the sections below, together with an analysis of their relevance and potential for tool integration on the desktop.

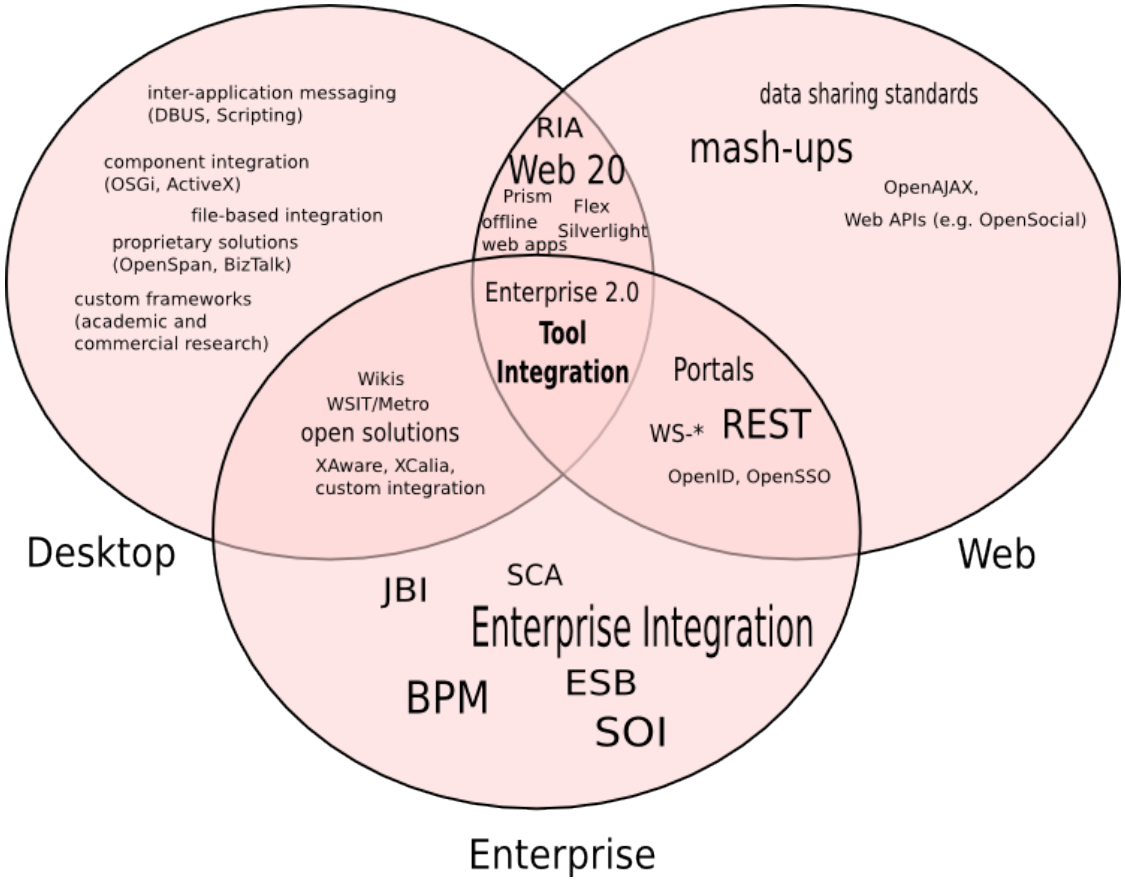


Figure 3.1: Integration solutions on the Desktop, the Web and in the Enterprise

3.2. Current Approaches on the Desktop

3.2.1. OS-Level Integration

This section discusses operating system-specific and therefore mainly proprietary integration concepts that can be used for native tool integration and for implementing higher level integration concepts on certain platforms. Although the solutions shown represent non-standard integration techniques tailored to specific platforms, they provide interesting examples of how integration can be achieved between desktop applications in a relatively *loosely coupled* way. This method of integration is regarded as the most stable (as mentioned in Section 2.6), as it allows replacing or modifying individual tools without breaking integration, which is a major requirement for tool integration (c.f. [Altheide2002]). Consequently, many characteristics of *OS-level* integration concepts can also be found in other areas, up to enterprise integration, which is a result of the ongoing *convergence* of the desktop, enterprise and web domain, as shown in Figure 3.1.

3.2.1.1. Integration on the file system level

As a form of *data integration*, common file formats allow transparent data exchange between applications, as long as a common format is defined and available to interested parties. On the desktop, common exchange formats have been used to a greater extent and range from CSV (comma separated values, used for tabular data and commonly supported in spreadsheet applications) to XML-based formats like XMI for modeling applications (as utilized in [Damm2000] for integrating modeling tools), X3D for 3d data, or the OpenDocument XML format ODF used in office applications. Of course, proprietary and closed binary formats are also widespread and make

it harder to integrate applications from different vendors. Here, frameworks like [ApachePOI] may be used to circumvent limited access to these formats.

Integration through common file formats is a useful approach for closely related applications that need to operate on the same data, but has several disadvantages: exchange of information is slow compared to message- or service-based information, as the receiver has to wait until the entire file is written. This makes *file-based* integration unsuitable for real-time environments where immediate user-feedback is needed, and hence less useful for general tool integration. Also, the quality of integration relies heavily on the quality of the file format: if it is ambiguously defined or lacks versioning information, exchange may fail because of data incompatibility. Lastly, this form of integration bears the danger of *semantic dissonance*, where data is interpreted differently across applications and contexts, e.g., numeric values may be incorrectly treated as absolute, relative or percent values [Trowbridge2004:57].

UNIX pipes are another example of *file system-based* integration, but here files are treated in a special way, following the *Pipes&Filters*-pattern: special files called *pipes* act as communication channels and allow transparent exchange of byte streams between two applications that want to communicate synchronously. This has been a very effective way to build *tool chains*, where tools, often small system programs like grep (the textual search tool) or awk (a string manipulation tool) are closely linked together and the receiver understands the format of the sender. Until now, C/C++ programmers utilize a chain of compiler tools for building applications, which still consists of separate command line tools such as gcc, configure and make that together realize a transparent build cycle. Now, with graphical IDEs and the widespread use of 4G languages such as Java or C#, these system-level tool chains are mostly restricted to kernel or driver development and embedded systems, but still a common way to handle system administration tasks.

One disadvantage of this integration pattern is that the content is simply exchanged as a binary byte stream or as raw text, but no information about the content is exchanged, so the semantics have to be interpreted by each participant individually, which results in redundancy and possible discrepancy among tools in a (piped) tool chain (as constituted by [Brown1992]). Pipes operate at the lowest level of integration, which is not enough for rich tool integration in the context of this work, but may function as a prerequisite for subsequent higher level integration (e.g., IDEs may offer a semantic layer of integration by interpreting the error output of a compiler tool chain, directing the user to the corresponding location in the source code).

Modern file systems offer higher-level data integration, which goes beyond simply using common file formats, by using common *metadata* and other shared semantics, and by offering methods to access the metadata from any application. The BeOS file system (BFS) provided a common abstraction for various file types by providing a MIME-type identification system and a standard set of attributes for each file type. This allows to easily exchange e-mails or contacts between software applications because the file's content is stored in a standard format, often plain text or a native format (an email-message conforming to RFC 822, or a PDF file), and additional information is accessible through file attributes by using the high-level C++ API (e.g., subject, address and sender of an e-mail, or the page count of a document):

> The power of attributes […] is that many programs can share information easily. Because access to attributes is uniform, the applications must agree on only the names of attributes. […] From the user's standpoint a single interface exists to information […]. [File] attributes provide an easy way to centralize storage of information […] and to do it in a way that facilitates sharing it between applications.
>
> —Dominic Giampaolo, *Practical File System Design with the Be File System*

Recent years brought more ways of high-level data integration using metadata that is automatically extracted from files by OS level services, e.g., the desktop search engine Spotlight[1] (integrated into MacOS X), or Nepomuk, an

[1] which has been designed, among others, by the creator of the Be File System, Dominic Giampaolo (see previous quote)

EU project to implement the vision of a *semantic desktop* on Linux [Groza2007], [Sauermann2008]. Nepomuk integrates into the KDE desktop and provides a framework based on RDF and implements several services for extracting various metadata from files into a searchable index. This makes inherent semantics accessible to other applications in a rich manner, and allows users to query for data based on the metadata and semantic relations gathered. The vision behind recent approaches is to no longer view files only as simple storage units, but as rich information sources: data is freed from the application that was used to create it so that users and other applications are able to recompose and correlate available information on a more abstract level.

While these approaches show great advancements in making available previously hidden and inaccessible information spread over various applications and files, they only provide a form of data integration, but do not provide any means to integrate at a higher level. Tool integration needs more than just inspecting files after the fact, but direct access to needed tool functionality via common interfaces, services or APIs for application developers (both tool creators and integration architects), facilitating cooperation among tools and enabling access from external applications. This can only be reached with higher level integration techniques detailed in subsequent sections.

3.2.1.2. Functional Integration and Scripting

Scripting languages currently experience a renaissance due to the dynamic nature of Web 2.0, where light-weight frameworks based on scripting languages have certain advantages, mainly time-to-market and easy extensibility, over established and more formal, often complex frameworks. A prime example is Ruby on Rails, which pioneered many Web 2.0 applications that have gained wide-spread adoption, e.g. Flickr or Basecamp. Enterprise Java (JEE) applications have been more complex to develop, at least initially, and so many Web 2.0 startups chose more rapid approaches, albeit these often do not scale very well (e.g. Twitter) and may lead to quickly implemented but hard to maintain applications that often lack a solid architecture.

In an effort to better support Web 2.0 development, and to combine the strengths of both worlds, Java 6 has gained support for integrating scripting languages through [JSR223], which allows Java code to be mixed with code written in scripting languages, e.g., for implementing the backend of a web application in Java EE and using Ruby for implementing the frontend, but also beyond web development, for supporting existing tool integration languages like Tcl/Tk, which is introduced in Section 3.2.2.1 below.

As mentioned in Section 2.3.4, using dynamic languages or scripting greatly facilitates tool integration, even more so when the tool to be integrated uses a wide-spread, common scripting language such as Python, which is used, e.g., in the 3d modeling application Blender. Only recently, common and standard scripting languages have seen increased use and support in software tools. Previously, applications mostly provided their own scripting language for writing in-application macros (e.g. MathScript or DXL).

To be of real use for tool integration however, scripting has to be available on a common base to every application by default, crossing tool boundaries and providing developers with established APIs or other techniques that facilitate building scripting support into own tools as an integral core feature, which consequently ensures that access to other tools is always possible in return. Fortunately, scripting support is now available in most operating systems in several ways: On Windows, VisualBasic[2] has become the *de facto* standard in scripting, at least for Microsoft applications like MS Office. In an effort to provide a real system-wide scripting facility, the Windows Scripting Host (WSH) was introduced but never widely used beyond system administration scripts. In the IBM world, REXX[3] was used as the default scripting language, which originally allowed cross-application scripting on IBM's AIX, OS/2 and zOS platforms, but has since become open source and is available for a variety of platforms such as Linux, Solaris and Windows in the form of Open Object REXX, an object-oriented extension

[2] in this form more precisely VisualBasic for Applications, VBA

[3] REXX is an acronym for “Restructured Extended Executor Language” and has been certified as ANSI standard ANSI X3.274-1996. It is now supported by the open Rexx Language Association [http://www.rexxla.org/]

to REXX. The language is both suitable for programming stand-alone scripts and for using as a macro language inside applications. On UNIX, Tcl/Tk (see Section 3.2.2.1) provides a powerful scripting language specifically tailored for controlling tools; another widely known language useful for this purpose is Lisp, which is strongly supported in Emacs, as shown in [Goose2000] .

Apple MacOS was the first desktop operating system to provide developers and end users with a system-wide default scripting environment: AppleScript uses English language elements and allows even end users with basic programming skills to control applications using scripts. The technical background and history of AppleScript is discussed in [Cook2007], who explains the motivation behind a system wide default scripting interface: “One benefit of a standard scripting platform is that applications can then be integrated with each other. This capability is important because users typically work with multiple applications at the same time.”. This is a simple but important observation and directly addresses the problems in tool integration, especially on the desktop. A simple example using application-specific terminology is shown in Example 3.1 below:

Example 3.1: A simple AppleScript that performs a calculation in Excel

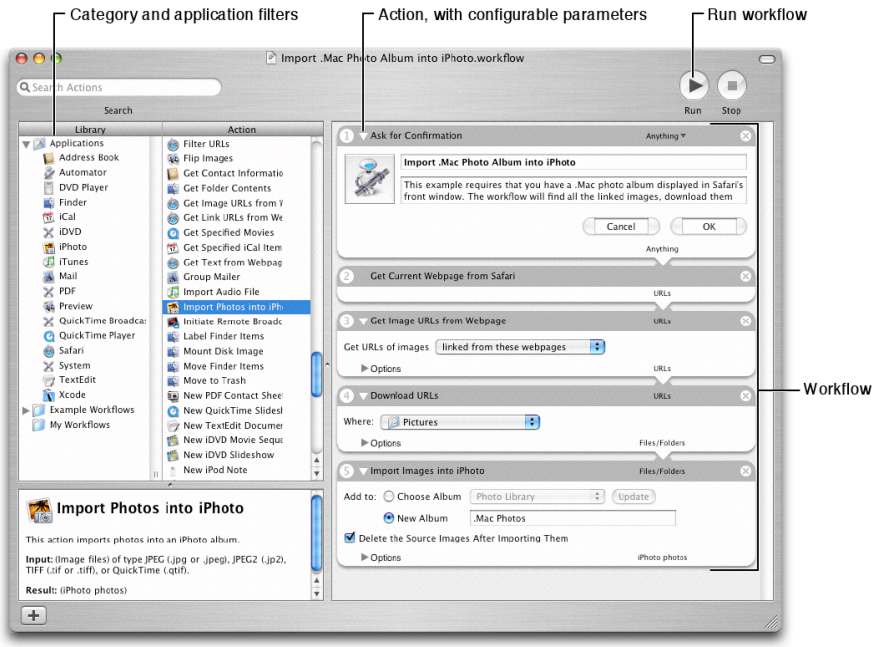
```
tell application “Excel”
   set formula of cell “A3” to “=A1+A2”
end tell
```

(from [Cook2007:17])

AppleScript is part of an *event-based* scripting API and enjoys widespread use in MacOS applications, which is important to make the scripting interface really useful for both developers and end users. For connecting scripting-enabled applications, the Apple Application Services-framework (formerly called Inter-Application Communication) provides an *application-level* integration approach using a common IPC layer, which also allows asynchronous and remote communication (also via SOAP[4]) and basic authentication. Applications are also connected on the user-interface level, albeit more simplistic, using a Services-menu: applications that want to take part in *Services*-integration[5] publish data types they support. When the user invokes the Services-menu, only *Services* that can handle the data types supported by the application are shown.

Scripting support is not bound to a specific language like AppleScript, as the OS provides a “standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages” through the Open Scripting Architecture (OSA, see [Cook2007]). This makes MacOS scripting support special as it goes beyond a simple default scripting language by providing several OS services and APIs that are part of a unique scripting architecture. Since MacOS X 10.4 (“Tiger”), there is also a visual designer for composing scripts, called Automator [Apple2007], allowing end users to build workflows for controlling applications through dynamically building scripts with application-provided *Actions*, as shown in Figure 3.2. This is in fact a rare example of user-oriented *process-integration* on the desktop.

[4]Consequently, in this context, AppleScript, precisely AppleEvents, are comparable to Web Services, c.f. [Cook2007:43]
[5]here, the term Services refers to the Services-menu, but it is also a simple form of service-oriented integration, see Section 3.3.3



(source: [Apple2007])
Figure 3.2: Automator allows visual process-integration of desktop applications on MacOS X

In a similar way, AppleScript Studio[6] allows advanced users or developers to construct new applications entirely from AppleScript-scripts.

On the Linux desktop, KDE4 now provides a uniform scripting platform for cross-application scripting: [KROSS], originally used in KOffice, integrates several scripting languages like Python, Ruby or JavaScript, and recently also Java and the lesser known Falcon language[7], by handling direct communication with scripting interpreters and providing a common, transparent API to applications. For adding scripting support to an existing application, the user's script is simply passed to the KROSS backend, which then dynamically invokes the relevant scripting interpreter.

It is worth noting that modern scripting interfaces are mostly realized through *message passing*, which is a concept commonly encountered in enterprise integration, as will be shown in Section 3.3.2. This shows another overlap between desktop and enterprise integration, as illustrated in Figure 3.1.

For general purpose tool integration, *scripting integration*, even if supported by the underlying OS, provides only a partial solution, as it leaves out many important aspects such as *end point abstraction* (by using logical endpoints that are resolved dynamically at runtime) and *data abstraction* (e.g., by using a more general, XML-based message format that can be inspected and enriched even by outside applications that do not understand all details of the transmitted content structure). Also, with the notable exception of AppleScript, support for security, distributed access, and asynchronous communication is not available in OS level scripting frameworks. However they do provide a common way for accessing tools and as such help integration developers in realizing this "last mile of tool integration".

[6] see the AppleScript Studio product site [http://www.apple.com/macosx/features/applescript/studio.html]
[7] see The Falcon programming language [http://www.falconpl.org/]

3.2.1.3. Application-Level integration

While scripting support through OS level APIs may be used to realize tool integration, full *application integration* goes one step further by enabling more fine-grained and far-reaching integration that is more transparent for users. Some solutions even provide an integrated user experience on the presentation-level by introducing a *component integration* model that divides applications into functional units that may be combined as needed. On the Windows platform, COM/OLE and later ActiveX are prime examples of application integration that reaches up to the user interface. On Linux, the KDE desktop offers a concept called KParts that provides similar functionality. For example, the Kontact PIM suite is just a container for independent e-mail, contact management, todo-list and notes-applications, and the HTML-renderer component KHTML may be embedded in other applications as needed, e.g., inside KMail for viewing HTML-formatted mail, or as part of the web browser Konqueror. KDE4 takes this concept even further and introduces a more *service-oriented* approach using KServices[8]. Another example which is more targeted at *information integration* are desktop Wikis like Tomboy[9], which allow connecting pieces of information in a semantic way using simple (textual) but "context aware" notes. References to documents and media files are handled externally by supporting applications (a simple but effective use of *launch integration*, see Section 2.3). Apple OpenDoc [Curbow1997] followed a similar vision, aiming at higher level information integration by dividing *compound documents* into interchangeable parts that are independent of the authoring application, providing abstraction of the common 1:1 binding between tools and "their" artifacts[10]. Similarly, Lotus Notes pioneered information integration between contacts, mail, "todo" items and other personal information, serving as an example for *application suites*, which can be defined as tool sets that "share data formats and operating conventions that let them meaningfully interact." [Brown1992]. Here, close cooperation between tools is only provided inside a fixed tool set, but this shows successful tool integration can provide true benefits to end users, making the new, interconnected "virtual" or meta tool more powerful than the sum of its parts.

A novel example for application integration possibilities on the desktop is the Linux Desktop Testing Project (LDTP), which uses existing accessibility libraries, scripting facilities and other features of the GNOME desktop environment for *user interface testing* and *automation* of GNOME applications. What would otherwise be a tedious undertaking now becomes feasible and effective, using scripting-based integration techniques (albeit constrained to a specific desktop environment).

3.2.1.3.1. D-BUS

Whereas OS-level inter-process communication is inherently platform-bound and often isolated inside local systems, D-BUS [DBUS] provides an integration infrastructure built on a general transport-layer for cross-platform, network-aware and protocol-agnostic inter-application communication[11]:

> D-BUS is an Inter-Process Communication (IPC) and Remote Procedure Calling (RPC) mechanism originally developed for Linux to replace existing and competing IPC solutions with one unified protocol.
>
> —from Introduction to D-BUS[12]

For interfacing with endpoints (i.e., tools), there are several types of bindings[13]: *language bindings* allow access to the bus from C, C++, Python, Ruby, Java or .NET applications, whereas *protocol bindings* allow communication over, e.g., `TCP/IP`, which extends the IPC layer by adding a remote interface. Furthermore, *interface*

[8] see KServices Tutorial [http://techbase.kde.org/Development/Tutorials/Services/Introduction]
[9] see the related article Tomboy and "Desktop Application Integration" [http://www.rahulgaitonde.org/2004/10/16/tomboy-and-thoughts-on-desktop-application-integration/]
[10] sadly, OpenDoc was cancelled and only its PostScript-engine survived in MacOS X
[11] coincidentially, DEC has developed a tool integration architecture with the same name, see [VanHorn1989]
[12] http://doc.trolltech.com/4.2/intro-to-dbus.html
[13] see the The D-BUS Bindings [http://www.freedesktop.org/wiki/Software/DBusBindings] page on the D-BUS Wiki

bindings exist for Qt (used in KDE) or `GLlib` (part of GNOME), providing *presentation integration* for various desktop environments. Lastly, D-BUS has been ported to other platforms such as `Windows` or `MacOS`, providing true cross-platform tool integration in an open way: the project has been incorporated into FreeDesktop.org as an open standard for inter-application communication, which is now used by KDE and the GNOME desktop, but also Enlightenment and other projects both in the commercial and open source world, even outside the desktop domain[14]. While D-BUS provides a lightweight, platform- and protocol-agnostic abstraction from IPC with supported applications, it does not provide the general integration infrastructure necessary for a tool integration framework, as it lacks more abstract concepts like mediation, transformation and reuse of integration components like Tool Adaptors. As a result, IPC-layers like D-BUS do not facilitate the realization of more general, high-level tool integration solutions[15]. A good introduction to D-BUS is provided in [Burton2004].

3.2.1.4. Summary

Looking at the disadvantages of OS-level solutions presented in this section reveals that most concepts are based on *component integration* (see also Section 3.2.3 below) and lack a more general API that abstracts integrated applications from specific environment properties or platforms. They offer only *tightly coupled* integration between related applications, and limit developers to concrete languages or platform APIs, which actually compromises the broader vision of application integration, as it does not work in heterogeneous environments.

Recent solutions allow more rapid and dynamic application level integration through visual interfaces and integration tools, but are limited to single platforms and target environments, e.g. OpenSpan, see Section 3.2.4.3.

3.2.2. Tool Integration Languages and Protocols

This section exemplarily lists two *languages-oriented* integration approaches: the first one represents a general-purpose *tool integration language*, whereas the second one is targeted at the Java platform. Both are cross-platform, but in a different way – Tcl has been ported to various platforms and offers extensions for integrating into target environments, whereas Java provides platform abstraction through the virtual machine.

3.2.2.1. Tcl/Tk

Tcl (short for *Tool Command Language*) is an early example (dating back to 1988) for a language specially targeted at solving *tool integration* problems [Ousterhout1994]. A short definition and motivation is given below:

> Tcl is an interpreter for a tool command language. It consists of a library package that is embedded in tools (such as editors, debuggers, etc.) as the basic command interpreter. […] Tcl is particularly attractive when integrated with the widget library of a window system: it increases the programmability of the widgets by providing mechanisms for variables, procedures, expressions, etc; it allows users to program both the appearance and the actions of widgets; and it offers a simple but powerful communication mechanism between interactive programs.
>
> —[Ousterhout1990]

This “tool command language” can be viewed as a *domain specific language* (DSL) for tool integration, like, e.g., SQL is for database access, but with a broader, more horizontal scope. A DSL can be defined as “a limited form of computer language designed for a specific class of problems.” [Fowler2005].

Originally targeted at controlling interactive `UNIX` command line tools, Tcl/Tk is now available on several platforms such as Linux, MacOS X and Windows, and has been extended in several ways[16]: The Tk extension, for

[14] see the D-BUS Projects [http://www.freedesktop.org/wiki/Software/DbusProjects] page on the D-BUS wiki

[15] The Eventuality [http://freedesktop.org/wiki/Software/eventuality] project aimed at implementing a more general purpose desktop integration framework on top of D-BUS, but has been discontinued.

[16] see Tcl/Tk Wiki [http://wiki.tcl.tk/940], which is part of the official Tcl/Tk web site

example, offers *presentation integration* by providing various GUI elements (widgets), whereas XOTcl[17] adds an object-oriented layer on top of Tcl. The language was used for successful tool integration solutions e.g., for the BOOST tool integration framework [Gautier1995], who concludes that “Tcl represents an approach towards providing a standard and quite powerful basis for programmability in software tools.”. With a tool-independent, standard language such as Tcl/Tk, it becomes possible for users to extend existing tools with new functionality (without having to learn tool-specific languages) that may be reused across tools, while at the same time transparently reusing functionality provided by tools themselves. This leads to a *service-oriented* view of tool integration, which was already envisioned in [Gautier1995]:

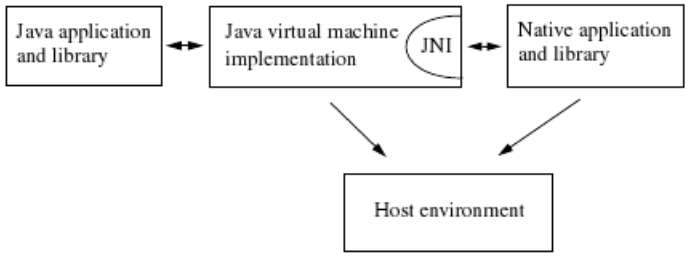
> We observe that this use of Tcl treats a tool as an object with operations invoked by a messages from elsewhere. Another way to look at this is that the tool is no more than a set of services invoked as required.

While a general tool control language is a useful concept, it comes with the prerequisite that the target tool has a Tcl-interface, i.e., it has to be linked against the Tcl library. This is an unacceptable restriction for *a-posteriori* tool integration, because existing tools often only provide custom interfaces and may have their own, proprietary *macro* language. Also, by default, Tcl provides only a C-based API, which makes accessing Tcl from other languages complicated and prohibits more modern approaches like object-oriented or service-oriented programming (however, basic *event-based* programming is possible through basic event loops).

For the graphical toolkit, Tk, there are bindings available to other languages such as Ruby, which would ease integration of graphical user interfaces into tools that support one of the scripting languages where there are Tk-bindings available. Lastly, with the vast amount of extensions also comes the problem of manageability and cross-platform support: many extensions are platform-specific and so may not be available for the target tool or environment. Nevertheless, Tcl/Tk still represents a proven and powerful approach to tool integration and with Tk, it is specifically targeted at desktop integration.

3.2.2.2. Java Native Interface (JNI)

JNI [Liang1999] provides low-level access to the Java VM for integrating C/C++-libraries and programs (called “native code”) by defining a standard method for bidirectional communication between both environments, as illustrated in Figure 3.3.



(source: [Liang1999:5])
Figure 3.3: JNI Overview

Developing a JNI interface involves several steps (c.f. [Liang1999:11]):

1. create a Java class that declares the native method

2. compile the program (using javac)

3. generate the header file (with javah)

[17] see XOTcl.org [http://www.xotcl.org/]

4. write the C implementation of the native method

5. compile C code and generate native library

JNI is best suited for solutions where certain portions of a program need to be optimized for speed, or where native code such as C libraries or applications are reused to minimize implementation costs or again to gain speed for critical portions of code. Because JNI is commonly used for integrating legacy code with Java, it was also used as part of the original ToolNet implementation, as described in Section 5.4.

As a drawback, JNI creates a *tight coupling* between the Java part and the native part, and introduces significant overhead for simple problems where only a single library has to be accessed and speed is not as important as flexibility and low maintenance cost. To address these issues, JNA, a relatively new library that wraps JNI, was used for realizing the prototype (see Section 6.4.1.3.3).

3.2.3. Component Based Integration Frameworks

"Dynamic Component Composition allows developers to stand on the shoulders of giants."
--Bill Joy on the Jini network technology, 1999

On the desktop, *component based software development* (introduced in Section 2.3.4.2) is a popular way to modularize applications and make them platform- and location independent. Several component frameworks are now available: The Netscape browser introduced [XPCOM] (short for Cross Platform Component Object Model), a cross-platform application framework which is also used as the underlying component model in the Firefox browser and its extensions, but also for stand-alone applications that use the functionality provided by available XPCOM libraries in a platform-neutral way, e.g. for realising networking, HTTP-communication, security, File I/O, web service access or rendering of web pages. Inter-component communication is provided via CORBA-like remote communication (using IDL, the CORBA interface definition standard) and there are various bindings that enable developers to develop XPCOM components or full applications using scripting languages such as Javascript, Python or Perl. Because of the network-centric API of XPCOM, general adoption beyond e-mail or web-applications is sparse. The API is also very complex and involves too much overhead for general-purpose tool integration.

Also UNO (Universal Network Objects) represents a platform-independent component model used in the OpenOffice productivity suite and associated plugins, which can be written in any language for which a UNO language binding (or bridge) exists, such as Java (using a JNI bridge), a .NET language (CLI bridge) or C/C ++. Tooling support is provided by IDEs, e.g., NetBeans supports the development of UNO-components (and hence OpenOffice plugins) in Java, .NET UNO components can be developed in VisualStudio. Although UNO could be theoretically used as a general, cross-platform component model that provides dynamic scripting access (using a VBA bridge or the native OpenOffice Script), it is closely modeled after the needs of an office suite and lacks more general-purpose platform features. Similarily to XPCOM, the API is grown and complex to work with, introducing a barrier for ad-hoc tool integration.

The Eclipse Rich Client Platform (RCP) [EclipseRCP] provides a modular tool platform for developing cross-platform, component-based Java applications with graphical user interfaces. The underlying component framework, Equinox [Eclipse2008], implements the now widely-used OSGi specification (see below), which provides the needed plugin-functionality and configuration management. Components (called Bundles in OSGi) can be dynamically installed, updated and removed[18], and component inter-dependencies or versioning conflicts are

[18]While the OSGi specification explicitly allows dynamic configuration, this is not fully implemented in Equinox, as it is very hard to realize without risking `ClassCastExceptions` when a component references a service that is not available anymore and forgets to refresh its dependencies. As a result, Eclipse recommends the user to restart the application when installing or removing plugins. This is also a heritage of previous Eclipse-versions (<3.0), where a proprietary component model was used that did not provide advanced configuration management features that are now supported by OSGi.

resolved by an intelligent classloading mechanism and automatic dependency management. The user interface is provided by the SWT toolkit, a set of standard UI components similar to Java Swing, but using native user interface controls on supported target platforms. A full definition of the RCP is given below:

> While the Eclipse platform is designed to serve as an open tools platform, it is architected so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform.
>
> —from the Eclipse RCP Wiki[19]

The Eclipse IDE is probably the best known implementation of an RCP-application, but several other tools and also domains outside software engineering are starting to utilize RCP, making use of common functionality and the modular foundation provided by the RCP framework, as shown in Section 3.2.4.2.1.

From this variety of custom frameworks, one plugin framework has evolved as a de facto standard, being openly developed by a cross-vendor organization and supported by an increasing number of projects (such as the previously mentioned RCP) and products: OSGi, which is covered in the next section. Prominent examples of component frameworks in the Java world are discussed in Section 3.2.3.2 below.

Lastly, there is even a component based *operating system* called ES[20], which is an effort to build an operating system that fully embraces a component based approach and integrates ECMAScript (the standardized JavaScript variant) at the system level, allowing access of both application and system components in a uniform way using IDL interfaces. This approach is promising since it provides scripts with full access to other applications and also to system level functionality, however for integrating existing tools this is not an option, as it depends on a single scripting language and operating system.

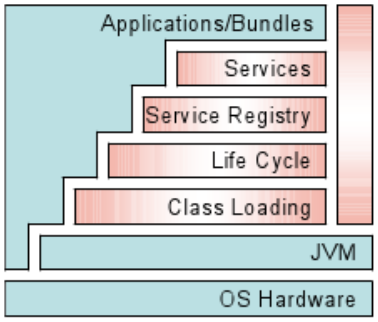
3.2.3.1. OSGi Service Platform

OSGi (specified in [OSGi2006], an overview is given in [OSGi2007]), stands for *Open Services Gateway Initiative* and "allows application programmers to develop small and loosely coupled components, which can adapt to the changing environment in real time. The platform operator uses these small components to compose larger systems."[21]. OSGi defines a component framework and related services and was originally targeted at embedded systems (e.g. TV settop boxes, but also automotive systems) to provide a unified, dynamic module system that handles dependencies between components that may have been developed by different vendors and allows on-the-fly reconfiguration and recomposition. This is in contrast to static component frameworks that require restarting or manual reconfiguration of the environment when a component is added or removed.

OSGi components, called *Bundles*, are simple JAR archives containing a standard manifest file with OSGi-specific headers that define provided and consumed *Packages* (exposed or required by other components). A common *service registry* handles publishing of and querying for services based on their public interface, which can then be used like a proxy. These three framework functions provide the basis for *service oriented* component integration, which is a very powerful but simple concept that can be applied very well to solve fragmentation, lack of reuse and poor application integration caused by version conflicts and dependency issues.

The underlying architecture of OSGi is shown in the following illustration, Figure 3.4.

[19] http://wiki.eclipse.org/index.php/Rich_Client_Platform
[20] see Introduction to the ES operating system [http://code.google.com/p/es-operating-system/wiki/XV_Semana_Informatica]
[21] from OSGi technical whitepaper [http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf]



(source: [OSGi2007:11])
Figure 3.4: OSGi architecture

OSGi became widely known in the software world when the Eclipse platform migrated from a proprietary component model to an open source implementation of the OSGi framework, called Equinox [Eclipse2008], which since then serves as the OSGi reference implementation; other open source implementations include [ApacheFelix] or [Knopflerfish]. The adoption of OSGi across the software development industry finally created a standard component model for desktop applications in the Java world, facilitating dynamic composition of service-based applications and tool platforms (see also Section 3.2.3.2 below). With several OSGi-based *component repositories*[22] available, a rich and open market place for interchangable components is currently emerging[23].

Moving beyond the desktop domain, also the enterprise domain is in the process of adopting OSGi as a component model, e.g., the Spring framework (see Section 3.2.3.2 below) has recently added support for OSGi components, and several application servers (Glassfish, BEA Weblogic, JBoss) are adopting OSGi as the underlying module system. The OSGi consortium formed an expert group[24] to cater for this emerging target domain.

Despite the benefits of a standardized and service-oriented component framework however, there are several limitations for applying OSGi as a tool integration framework, as it is targeted at a too low level for general purpose integration. The main focus is on the component model, not on a general, high-level infrastructure with reusable integration services or advanced, network-transparent message routing. Also, the manifest format is very limiting and does not provide enough information for tool integration; using a well established standard interface definition for services such as *WSDL* would have been preferable[25]. Although OSGi provides a Services concept, it only supports OSGi services provided by components, Web services are not supported. Also, the OSGi deployment model is currently limited to a single runtime, although efforts to make OSGi distributed are underway (c.f. [Jahn2008]), as part of Enterprise OSGi, see also project Corona in Section 3.2.4.2.1).

OSGi can therefore be used as a solid foundation to build a higher level integration framework that applies the concepts and solutions and adds necessary framework parts, as demonstrated by [Coalevo], an open source, OSGi-based service-oriented collaboration framework: Coalevo introduces *Protocol Service Bundles* to bridge communication protocols (e.g. HTTP, SSH), and *Protocol Adapter* Bundles to mediate between protocol and application services (e.g., presence, messaging, or user data). While the implementation is only in an early stage, the concepts are very similar to the *Java Business Integration* standard (see Section 4.2), which provides corresponding *BindingComponents* and `ServiceEngines`, respectively.

[22] see OSGi Bundle Repository (ORB) [http://www.osgi.org/Repository/HomePage], Eclipse Orbit [http://www.eclipse.org/orbit/] and recently the SpringSource Enterprise Bundle Repository [http://www.springsource.com/repository/app/]
[23] This could well be the component market as envisioned in *Component Software* [Szyperski2002], chapter 2.
[24] see The OSGi EEG home [http://www.osgi.org/EEG/HomePage]
[25] while this is currently being investigated for inclusion into a later revision, it is already supported in e.g. Java Business Integration, which is covered in Chapter 4, but comes with it own problems

3.2.3.2. Java Component Frameworks

In the Java world, components (or *Beans*) are the standard building blocks for applications and application resources. They are runnable (in Java SE) or deployable (in Java EE) artifacts that adhere to component standards and related contracts, specified through several JSRs. Examples include simple JARs for applications and libraries, JNLP (Java Network Launching Protocol)-packages for deployment over web browsers (termed "Java WebStart"), JMX MBeans used for management access, Enterprise Java Beans (EJBs) for web or enterprise applications and JCA Adaptors specified by the *Java Connector Architecture* (see Section 3.3.7.1) packaged as *Resource Adapter Archives* (RARs).

In the same way, these concepts are also examples for component integration: from functional integration in JARs to service-oriented integration in EARs or enterprise integration in RARs. Some solutions are more tightly coupled (like communication between Enterprise Java Beans over RMI), while others allow loose coupling (e.g., accessing JMX `MBeans` using a web interface).

JMX reaches even further by integrating applications into management consoles, crossing network and protocol boundaries: management standards such as SNMP are supported, allowing administrators to connect to managed applications using existing, commercial off-the-shelf management consoles such as HP OpenView, but also open source solutions such as OpenNMS[26]. Also HTTP and other bindings are provided, allowing web based access to managed applications.

JMX combines several forms of integration in a transparent, uniform API: *presentation integration* in the management interface provides a combined user interface for controlling and inspecting managed applications (which provide parts of the user interface), *protocol integration* through JMX *Protocol Adapters* bridges different access methods and standards, such as HTTP, SNMP, WBEM[27] or IIOP. On a higher level, JMX realizes *application integration* by connecting managed applications to a central management application. The implementation is realized through *component integration*, using JMX MBeans to provide a management façade for the managed application.

Besides official standards, open source plugin frameworks such as the JavaPluginFramework [JPF] or OpenAdaptor (see Section 3.2.4.1), which use existing Java component standards and provide additional integration services such as pipes and also Adapters for integrating with external protocols and existing applications or other facilities such as message queues.

In the Java enterprise domain, there has been a strong trend in recent years towards to more dynamic and open component frameworks such as Spring and OSGi (see previous section), complementing or replacing well established standards defined in the JEE specification. This movement has been spearheaded by the Spring framework [Spring], an open source Java component framework. Spring realizes lightweight component integration through *dependency injection* (also called *Inversion of Control*[28]), a concept which eliminates the need for hardcoding references to data sources and other artifacts such as JCA Adapter configurations or database configurations, by using XML configuration or metadata (Java Annotations). At runtime, the framework inspects the configuration, resolves all references contained, and "injects" the targets into proxy (or placeholder) objects provided by the application, using Java Reflection.

While dependency injection alone is not enough for tool integration, it provides an essential foundation and allows connecting external sources dynamically at runtime through a loosely coupled component model. On

[26] see the OpenNMS.org site [http://www.opennms.org/], which also provides an example for application integration using component integration and scripting, which is described in the white paper Hyperic Integration [http://www.opennms.org/images/3/3a/Hyperic-integration3.pdf]

[27] for *Web-Based Enterprise Management*, a management standard created by the Distributed Management Task Force (DMTF), see the DMTF WBEM page [http://www.dmtf.org/standards/wbem/]

[28] this concept was first introduced by Kent Beck in his essay "Inversion of Control Containers and the Dependency Injection pattern [http://martinfowler.com/articles/injection.html]"

this basis, external tools could be integrated as needed by changing the configuration, but in practice dynamic reconfiguration does not work because of Java classloading restrictions that make it almost impossible to add or remove modules at runtime. Also, the Spring framework has lacked service oriented concepts (see Section 3.3.3 below) and relied on Java `Beans` as the underlying component framework. Lastly, there are no integration facilities like Adapters or Translators, and there is no declarative way to specify the desired integration structure.

These shortcomings have recently been addressed by the Spring Dynamic Modules for OSGi project [SpringDM] and [SpringIntegration] (see Section 3.2.4.1 below). The former uses OSGi as the component model, but hides the complexity involved in creating OSGi bundles and manifests with explicit references. By using a dynamic component framework like OSGi, it solves classloading problems, enabling truly dynamic solutions which allow installation, starting, stopping and removing of components at runtime. This allows the addition and reconfiguration of Tool Adapters as needed, which is an important requirement in adaptive integration solutions. Higher-level integration concepts and patterns are implemented with Spring Integration, which is covered in Section 3.2.4.1 below.

In order to retrofit current independent Java component frameworks that have become de-facto standards back into the broader JEE-standard, several new JSRs have been issued, the most important are JSR 277, which defines the API and deployment specifications for a dynamic Java module standard in a more static way, and JSR 291, which directly integrates the OSGi component specification into the core platform and also covers the dynamic aspects of the component framework. Both JSRs are planned for inclusion into Java 7, which is to be released in the first half of 2009[29].

3.2.4. Current Tool Integration Solutions on the Desktop

This section provides a compact overview of currently available tool integration frameworks, which combine several patterns and concepts presented above to provide a basis for concrete tool integration solutions, which are covered as well.

3.2.4.1. Open Source Solutions

"Modularization of code has made software and its component parts more interchangeable, and created opportunities for niche market players to reassemble components to make new products and services."
--from [Samuelson2006]

OpenAdaptor [OpenAdaptor2007], [Lachor2008] is a typical component integration framework for Java, as mentioned in Section 3.2.3.2 above. The lightweight enterprise integration-framework can also be used to integrate desktop applications. `Connectors` realize protocol bindings for connecting input sources and output sinks (e.g., protocols like HTTP, FTP, JMS, SOAP or simple files), whereas `Processors` translate between different data formats (XML, CSV, JDBC ResultSets). The resulting solution acts as a pipeline that realizes the desired integration functionality.

The framework has been used in commercial settings for various enterprise integration projects and has been recently rearchitected, using Java Beans as the underlying component model and Spring for configuration. Unfortunately, Java Beans lack more sophisticated mechanisms for specifying dependencies and they do not support a dynamic lifecycle (see above). Consequently, most frameworks and solutions, including JEE application servers, are currently migrating to OSGi as the underlying component model, which provides rich support for dependencies and dynamic deployment, undeployment, start and restarting of components. Additionally, OSGi provides a service-oriented programming model is available and offers dynamic provision, search and consump-

[29] for a good overview of the current status of Java component standards form past to present, including JSR 277 and JSR 291, see the article "The case for Java modularity [http://www.javaworld.com/javaworld/jw-08-2008/jw-08-java-modularity.html]" (JavaWorld 08/2008)

tion of Services at runtime. These aspects are missing from OpenAdaptor, which only supports hard-wired, static configurations, where pipelines cannot be changed at runtime. Integration is only supported at the data and functional level, and routing has to be implemented in custom Adaptors. There is also no tooling available, so integration has to be done by hand and through manual *XML* configuration and coding.

Spring Dynamic Modules for OSGi [SpringDM] is a modern component integration framework for Java EE that uses OSGi as the component model and provides a façade to JEE developers for transparent integration with existing Spring configurations and other Java EE frameworks and standards (e.g. JPA and persistence libraries). Closely related but targeted at higher level integration is the emerging Spring Integration [SpringIntegration] project, which realizes the integration patterns found in [EIP] by providing various *Adapters* (e.g. File, remote messaging) that can be configured via common Spring configuration mechanisms (such as XML files). Custom Adapters can be realized as simple Java Beans that are later connected through configuration. Although this concept enables loosely coupled integration solutions, there is also the danger of increasing complexity and scalability problems as integration scenarios get more sophisticated, which is a major argument for *model-driven* integration, see Section 3.3.6. It has to be seen when and how Spring Integration will be integrated into the common Spring tooling platform, which is available as an Eclipse plugin.

[Apatar] is a visual *data integration* solution that supports the *extract, transform and load* (ETL)-pattern and offers connectors for common databases and third party legacy applications. The solution is open source and cross-platform (realized in Java), and the designer is based on the Eclipse platform. Integration architects build so called data maps using familiar design concepts (endpoints represent applications, which are connected via edges to *Transformer* components). The resulting data maps can then be embedded as a server-side application or directly incorporated into custom solutions. The community site also acts as a repository where existing solutions can be reused and searched for, and new contributions may be shared. Although Apatar may be used for application integration on the desktop, like OpenAdaptor, because it is relatively lightweight and can be embedded into custom applications, it is mainly targeted at enterprise integration, as covered in Section 3.3.3.3.1.

The remaining open source solutions mainly build on Eclipse and associated integration and modeling facilities, which is covered in Section 3.2.4.2.1 below.

3.2.4.2. Eclipse as an Integration Platform

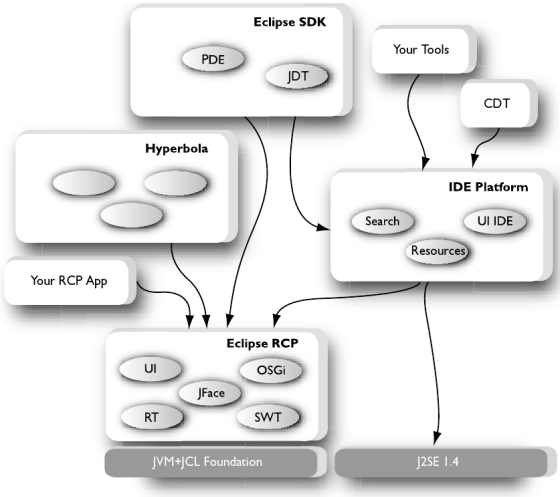
> Eclipse is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle.

The [Eclipse] project is an open source development platform based on a plugin-based core, Equinox [Eclipse2008], which is the reference implementation of the *OSGi (Open Services Gateway Initiative)*-specification (specifically, the framework-part) as defined in [OSGI]. The specification defines a framework for component-based systems, which are entirely based on reusable, loosely coupled modules (called *Plugins* in Eclipse and *Bundles* in OSGi). Plugins are components that follow a common contract (specified by an XML configuration, the plugin manifest) for defining dependencies on other plugins, and for exposing the own functionality for reuse inside the framework. This information is separated out from the plugin implementation so that dynamic discovery of dependent plugins is possible on demand, when a plugin is loaded into memory. OSGi concentrates on deployment, discovery and lifecycle-management and allows dynamic reconfiguration, e.g., installing or removing plugins, without having to restart the runtime.

The minimal set of plugins required to run an Eclipse-based application, including the required Java Runtime Environment (JRE), form the *Rich Client Platform (RCP)*, which enables the adaption of Eclipse for a wide array of applications, far beyond the well known Java IDE, which is also realized through a set of plugins, the *Java Development Tools* (JDT). This platform can be used as a foundation for plugin-based applications that run independently of Eclipse and are tailored to custom requirements and projects, while still being able to reuse

existing plugins as part of the individual solution. Viewed from this perspective, the Eclipse Java IDE itself is just a specialized RCP application targeted at software development, using a coherent set of development-oriented plugins (the JDT and related extensions such as the *Web Tools Platform* (WTP) for JEE development).

Figure 3.5 shows an overview of the RCP architecture, denoting the essential plugins needed for an Eclipse application-runtime.



(source: [EclipseRCP:14])
Figure 3.5: Eclipse RCP architecture overview

Examples for solutions that use the RCP include the IBM Rational product line, Lotus Notes, NASA Maestro (a solution to remote-control space vehicles), business reporting and workflow systems (using Eclipse BIRT), or solutions for healthcare, and of course ToolNet (see Chapter 5).

The RCP is also moving towards the embedded domain with [eRCP], which allows to build RCP applications for mobile and embedded devices (it is already supported by Nokia's S60 OS). At the same time, the RCP is moving towards the web and server-side applications with the *Rich Ajax Platform* (RAP) [EclipseRAP], following the current trends in desktop and web convergence. RAP allows developing rich web applications based on the Eclipse plugin model and the SWT UI toolkit: “RAP is very similar to Eclipse RCP, but instead of being executed on a desktop computer RAP is run on a server and clients can access the application with standard browsers. This is mainly achieved by providing a special implementation of SWT (a subset of SWT API).”[31].

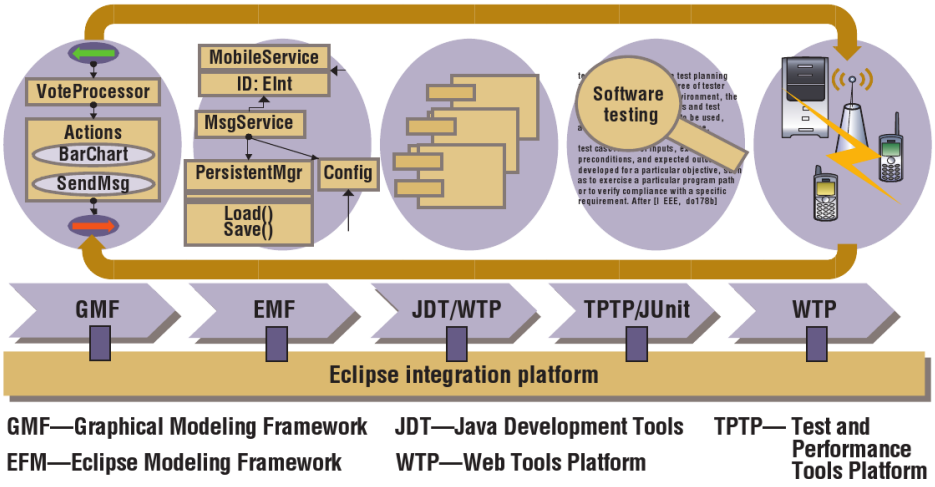
[EclipseRCP] provides detailed coverage on developing RCP-based applications, while [Eclipse2006] gives a good introduction to Eclipse in general and the RCP in particular.

3.2.4.2.1. Current Tool Integration Solutions based on Eclipse

A more modern and scalable approach is to create a multi-layered interoperability framework leveraging SOA technologies. Tools can be orchestrated to provide repeatable, efficient processes that are responsive to changes in business needs by building upon an interoperable and collaborative collection of services and components. The tool provider can expose as much (or as little) as they choose and the consumer of these technologies will have the ultimate control over how these technologies are orchestrated together.
--Eclipse Whitepaper Integration and Interoperability of Application Lifecycle Management tools

[31]from the project homepage [http://www.eclipse.org/rap/about.php]

Eclipse provides a component-based framework for integrating software tools into a common working environment, as proposed in [Yang2007]: “Fundamentally, Eclipse is a framework for plug-ins. [...] Other tools plug into this basic framework to create a usable application. Plug-ins add functionality through predefined extension points that the Eclipse platform offers.”. This concept is illustrated in Figure 3.6 below:



(from [Yang2007])
Figure 3.6: Eclipse as a Tool Integration Platform

By using a common component model (OSGi), tools are realized as components and connected in a loosely-coupled way by publishing *extension points* that other tool components can reuse and extend further. The OSGi-based component model, which constitutes the *Eclipse Rich Client Platform* (RCP), was introduced in Section 3.2.3.

Although “Eclipse simplifies tool integration by allowing tools to integrate with the platform instead of each other.” [Amsden2001], it only provides the base platform to build such a solution, but does not include needed tool integration facilities like *Tool Adapters*, data translators or a common communication infrastructure like a *message bus*. This is also noted in [OMG2004]: “[…] tool coordination frameworks, such as Eclipse, provide tool chain integration for data and control flows, but do not take into account semantic integration issues. New standards and facilities for formally representing and transforming tool data are required.” (one emerging standard is JBI, which will be covered as part of the proposed solution in Section 4.2).

This section therefore presents relevant tool integration solutions that build on the Eclipse framework but add advanced tool integration functionality.

With the exception of the ToolNet-framework [ToolNet], which is covered separately in Chapter 5, there are very few Eclipse-based solutions suitable for general tool integration. Most existing solutions focus on *model-based* integration (see Section 3.3.6), building on top of the *Eclipse Modeling Project*[32] and its sub projects, such as EMF (Eclipse Modeling Framework), GMF (Graphical Modeling Framework) or related framework projects for model transformation and UML modeling. An example for such a solution is [OpenArchitectureWare], which can be shortly defined as “a tool for building model-driven tools.”. The open source solution implements comprehensive tool support for metamodeling based on the EMF and provides a rich template language (*Xpand*) for complex code generation.

TOPCASED (Toolkit In OPen source for Critical Applications and SystEms Development, [TOPCASED2008]) is a project to create an open CASE platform for engineering (mainly automotive and aeronautic engineering,

[32]see Eclipse Modeling Project [http://www.eclipse.org/modeling/]

project members include SiemensVDO and also EADS), based on open source technologies and formal methods. The focus lies on ensuring long-term availability (hence the requirement for open source modules) and reliability, both due to the target domains. Engineering tools are integrated via meta-models (using the Eclipse ECORE meta-modeling language), and so-called “meta-tools” capture common configuration and functionality shared between tools. Various proprietary technologies are exchanged with custom built open source solutions, e.g. for requirements tracing or validation. TOPCASED embraces the Eclipse infrastructure and reuses existing projects where possible. Client tools (either TOPCASED editors or proprietary tools) are connected to a service-oriented bus that is actually a set of Eclipse-plugins providing common infrastructure services. Remote tools are connected via a SOAP Adapter. The main goal is to provide a reliable platform for systems development, as such it is rather static and not aimed at dynamically integrating tools (before adding new tools, a formal validation is required).

The *Open System Engineering Environment* (OSEE) is an Eclipse-project that “provides a tightly integrated environment that supports lean engineering. It is integrated around a simple, user-definable data model to eloquently provide bidirectional traceability across the full product life-cycle […]” (from the project homepage[33]). The project aims to support the V-model, but is still in incubation phase. Like TOPCASED, it is more of a static development platform that is not suited for dynamic, user-centric tool integration with a focus on transparent tool-to-tool integration and user-interface integration.

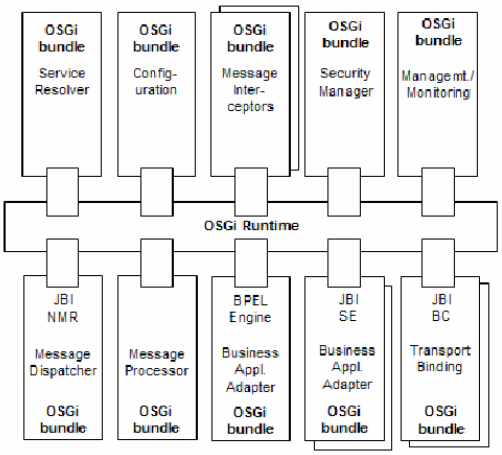
The aforementioned projects show that Eclipse can provide a useful basis for building CASE platforms, but for more general tool integration, two recent Eclipse-projects are of particular interest: Corona is a tool services framework similar to an *Enterprise Service Bus* (see Section 3.3.3.2) that connects distributed Eclipse instances, providing *location transparency* for Eclipse-based tools. This allows remote tool collaboration between client and server based tools. The Application Lifecycle Framework ALF realizes an event-driven workflow integration of tools on top of a common infrastructure, enabling *tool orchestratio*n from an integrated business process, where events are routed between collaborating tools. Together, these frameworks provide a way to build integrated processes and tool chains, as described in [Parker2006].

While a combination of ALF and Corona provides many features required for a general tool integration framework, but lacks facilities or standards for integrating external components that are not built for the Eclipse platform – tools are assumed to be OSGi components; for existing tools, Adapters have to be provided. Also, the communication backbone in Corona adds some ESB services missing in the standard Eclipse platform, but leaves many others up to developers, e.g., message translation, advanced routing and many other enterprise integration patterns found in [EIP]. For remote communication, only web services are provided, which has shown to be problematic for loosely-coupled and dynamic integration in Section 3.3.3.1. Lastly, there is no common concept for integrating existing, non-Eclipse tools using standards like JCA or other Adapters. This Eclipse or OSGi centric view does not reflect the rich and heterogeneous tool landscape encountered in the target domain of this worok.

The *Eclipse SOA Tools Platform* (STP) [EclipseSTP2006], [Mos2008] is targeted at *service-oriented integration* (see Section 3.3.3) and provides tool support for building composite applications for an SOA environment. The solution consequently applies a model-based approach throughout the process, using model-transformation to adapt to different models encountered in the SOA world, such as the *Service Component Architecture* (*SCA*)-standard. Support for *Java Business Integration* (JBI) is planned for a later stage. In the meantime, ChainBuilderIDE and the NetBeans CASA editor provide visual SOA tooling for JBI, as shown in Section 4.2.3.

Project Swordfish [Swordfish] builds on the aforementioned SOA Tools Platform and provides an accompanying runtime framework for an SOA, based on OSGi and integrating standards like JBI (using the ServiceMix kernel), which is used to integrate with existing business applications over Adapters and for integration into BPEL processes, and SCA, which is used for describing and packaging composite services. The architecture resembles an OSGi-based ESB, as shown in Figure 3.7 below:

[33]see The Open System Engineering Environment Homepage [http://www.eclipse.org/osee/]



(from , [Swordfish], Eclipse creation review)
Figure 3.7: Architectural overview of Project Swordfish

With the exception of the emerging Project Swordfish[34], Eclipse based solutions tend to emphasize on the (centralized) *orchestration* of tools, whereas this work seeks to realize a framework that provides (decentralized) *choreography* of tools, allowing tools working together in a *peer-to-peer* fashion. The “Eclipse way” is more about integrating tools into a central workplace, whereas tool integration in this work is more about connecting tools as-is in a service-oriented and user-centric way, sharing data and functionality, but at the same time staying within original tools and providing a rich integrated user experience.

3.2.4.3. Commercial Solutions

While a multitude of enterprise integration solution exists today, there are still few examples of integration solutions that specifically target the desktop. This section presents two similar solutions that use an approach called composite service integration (CSI) or client-side integration (as opposed to enterprise- or server-side integration), which is different from the other integration solutions presented in this chapter.

[OpenSpan2008] is a closed commercial solution that has been designed to solve integration problems on the Windows/.NET platform and has already been successfully deployed in several projects, e.g., in the call center domain. *OpenSpan* dynamically inspects target applications to be integrated and exposes the application's objects and methods as building blocks and services that can later be combined to new, composite applications. OpenSpan provides so-called *Integrators* for several application types, from main frame (*green screen* applications) applications over Java applications to typical Win32-applications. Also web services are supported, and a major vision of OpenSpan is to connect legacy applications to the SOA world, allowing communication between existing applications with new, web service based applications, allowing to create mashups that reach beyond the web, into the desktop. For integration architects, a visual designer is provided that allows inspecting the target application's interface for needed parts (e.g., text fields) and functionality (e.g. a "Send" button), which are then automatically extracted and exposed for later reuse in the composite application designer.

AppIntegrator[35] uses a similar approach and has been used for integrating applications in several domains such as healthcare, content management, education, or in insurance companies. Additionally, document management capabilities can be added through a separate integration product, DocConnector, which interfaces with existing document and content management systems.

[34] planned for release in October 2008
[35] see the company site Karoroa.com [http://www.karora.com/appconnector/appconnoview.htm]

While these products provide a compelling way for rich application integration on the desktop and into the enterprise and web domain, the costs and dangers of those seamingly easy and powerful integration approaches are manifold: They are closed and proprietary, as there is no uniform, standards-based description of the applications' interfaces, so the resulting solutions are bound to a single integration product, bearing the danger of vendor lock-in; also new types of applications (e.g., scripts and applications realized with dynamic languages) cannot be added without vendor support. The resulting integration is mostly static, as new components cannot be added on the fly without rearchitecting and redeploying the entire solution. As these solutions integrate mostly at the method and interface level, there is a tight coupling between applications, where methods and services are called directly, and user interface elements are accessed by their object names, rather than using dynamic lookup or logical names as in service-oriented applications. Lastly, the proposed visual-centric integration approach may result in “quick-and-dirty”, grown solutions that simply perform point-to-point integration, which is an anti-pattern because of bad scalability, maintenance, and adaptability.

3.2.4.4. Tool Integration in other domains

Although this thesis focuses on tool integration in software engineering, it is worth noting that also other areas have seen the need for tool integration and some standards have evolved, especially in media creation, where it is very common to utilize a wealth of highly specialized tools for working on the many aspects of media creation.

[Verse] is an example for a successful tool integration standard in the digital media and computer graphics or game creation domain, see [Brink2001]:

> Normally the content, tools and rendering technology are very tightly interlocked. The engine can only take specific data that is made with very specific tools. By separating the three we create a much more dynamic pipeline where the content is stored in Verse format that is not looked to one specific rendering technology or tool.
>
> —from the Verse homepage[36]

The standard defines a common low-level network protocol, 3d data format and repository for artifacts or assets that artists create during their work, such as models, textures, scripts or audio and video data. Artists can work on the same data, which is connected through data linking, in a distributed environment, which is not uncommon in the media creation space. Several 3d programs, like Blender, 3D Studio MAX or The GIMP already support the Verse protocol. Verse is however not suitable for use in a more general tool integration context, as it is clearly targeted at a specific domain. While it abstracts from individual tool formats and APIs, Verse introduces tight coupling through the use of method-level control integration (e.g., via callback functions). However, the Verse vision shows the potential of tool integration and the mutual benefit that comes from bridging disparate tools and data.

[AutoSAR] (short for “AUTomotive Open System ARchitecture”) is a standards effort in the automotive domain that “defines a standardized component model consisting of a clear programming language mapping (syntactically) and a file format for component requirement and capability description.” (from the specification). Automotive applications are composed of components that are connected over a “Virtual Function Bus” (VFB), which can be seen as the logical layer above hardware bus systems such as LIN or CAN, which are common in vehicle systems. The standard “supports the way towards an integrated and tightly coupled tool chain for automotive software development.”. Interestingly, the standard does not mention the OSGi component framework (introduced in Section 3.2.3.1), which has a long history in the embedded domain, whereas AutoSAR's component model seems to be completely independent and proprietary. OSGi, on the other hand, is investigating an implementation targeted at the automotive domain through its OSGi Vehicle Expert Group[37], as major market players such as BMW expressed interest in adopting OSGi. Naturally, the same critique is true for this standard

[36] http://www.quelsolaar.com/verse/pipeline.html
[37] see the web site at OSGi Vehicle Expert Group [http://www.osgi.org/Vehicle/HomePage]

as for other component centric standards, with the exception that OSGi is moving towards the enterprise space and provides a rich Java API including service oriented concepts that make it a better candidate for using it as a base for a tool integration solution.

3.3. Related Approaches in Enterprise Integration

"[…] you need a logical design at the integration level, just like you need a logical design at the application level."
--[Trowbridge2004]

The first part of this chapter has shown various integration concepts and solutions targeted at the desktop. From this analysis, it can be concluded that a general tool integration framework needs to incorporate higher-level standards and solutions in order to provide a more dynamic and general solution. In the enterprise domain, there is a similar need for a reusable integration infrastructure, and several patterns and solutions are already available.

While earlier enterprise integration efforts were limited to custom built point-to-point integration solutions or vendor-specific proprietary middleware, there are now many higher level solutions available, building on established open standards that evolved from practical experience gained through integration projects in the industry. This section covers concepts, patterns and best practices collected through research and through landmark work like [PofEAA], which concentrates on designing and implementing service-oriented applications, and [EIP], which introduces a common set of patterns that describe how to integrate applications using messaging and related concepts.

3.3.1. Definitions

Before continuing with a survey on concepts in enterprise integration that are important for understanding the proposed solution, the general context of enterprise integration and associated terms are defined. The remainder of this chapter presents concrete solutions that are adapted and applied to solving the problem of desktop tool integration in Chapter 4.

First, a *service-oriented architecture* (SOA) is the currently most widespread design paradigm in the enterprise domain, with the key principle of "encapsulating application logic within services that interact via a common communications protocol." [Erl2004:51]. Web Services are a common way to implement an SOA, but there are other ways to adopt service-oriented principles, as shown in Section 3.3.3 below. In an SOA, service interfaces are defined using the *WSDL* standard and messages are exchanged in XML-format, as specified by the *SOAP* protocol standard, usually over HTTP[38].

Enterprise integration, according to [EIP:39], "is the task of making disparate applications work together to produce a unified set of functionality.". If we replace "applications" with "tools", it becomes clear that *tool integration* has much in common with enterprise integration, thus it is worth examining exiting solutions in enterprise integration for applicability to (desktop) tool integration.

The general term *middleware* has been used for various enterprise integration solutions, which have previously been mostly *message-oriented* and consequently been called *message-oriented middleware* (MOM), which is defined in [ESB:77] as "a concept that involves the passing of data between applications using a communication channel that carries self-contained units of information (messages).". The recent shift towards SOA has resulted in more open and flexible middleware solutions, which is reflected in the definition given in [Schmidt2006]: "In SOA middleware, software components provide reusable services to a range of application domains, which are then composed into domain-specific assemblies for application (re)use.". JEE, .NET, and the (now largely

[38]see also Thomas Erl's web site whatissoa.com [http://www.whatissoa.com/]

obsolete) CORBA Component Model (CCM) are then given as example. In today's integrated enterprise world, this is a very generic definition, and the given examples are growing into application platforms that are integrated themselves, using a modern integration infrastructure such as the *Enterprise Service Bus* (see Section 3.3.3.2), which can then be viewed as a “middleware for middleware technologies” (c.f. [Juric2007]).

Recently, because of the complexity and infrastructural demands of the SOAP-protocol typically used in an SOA, REST [Fielding2000] has emerged as a lightweight alternative and is increasingly used in small to medium-sized web applications, forming a new paradigm of *resource-oriented computing*, see Section 8.5.

But there is more to SOA than a set of technical standards, as noted in [OASIS2008:10]: “From a holistic perspective, a SOA-based system is a network of independent services, machines, the people who operate, affect, use, and govern those services as well as the suppliers of equipment and personnel to these people and services.” (c.f. [Erl2004:476], who provides a similar view of the far-reaching scope of SOA).

In a similar way, the concepts presented below mostly work hand in hand, and integration solutions usually apply a mix of several techniques, because, as will be demonstrated below, each approach has advantages and weaknesses. It is therefore necessary to evaluate current best-of-breed enterprise integration solutions and patterns from server-side environments for applicability to client-side tool integration. Although commercial solutions are not an option in the context of this thesis, which proposes an open tool integration platform, they are briefly covered as a reference and evaluation of useful concepts.

3.3.2. Message Based Integration

Whereas previous approaches to enterprise integration often used synchronous, functional integration using RPC-style method invocation or proprietary message-oriented middleware (see Section 2.6.1), modern *message-based* integration solutions are based on common, open message formats (often standardized and XML-based) for application-neutral message-exchange. A *message* is defined as “an atomic packet of data that can be transmitted on a channel.”, where a message *channel* is “a virtual pipe that connects a sender to a receiver” [EIP:57].

Current solutions include APIs like JEE's Java Message Service ([JMS]), which implements a *message queue* for realizing transparent messaging between Enterprise Java applications, as shown in [EIP:187]. The message queue is responsible for connecting message senders to receivers and for ensuring reliable transmission of messages, even when the receiver is not available all the time. Applications can send and receive messages *synchronously* (e.g., by implementing the *request-reply* pattern, see [EIP:154]), or *asynchronously* by subscribing to messages of interest and sending messages to the message queue without waiting for a response, following the *publish-subscribe* pattern [EIP:106], which is an implementation of the *Observer*-pattern ([GoF:293]). When a message of interest arrives, a callback method is called on the subscriber. If the receiver is not available, the message is *queued* and delivered as soon as the receiver becomes available again.

Using messages for integration provides many benefits and possibilities such as *message inspection* (e.g., for ensuring the existence of a session token), message *enrichment* (e.g., adding metadata), or message *transformation* (e.g., between proprietary formats). Message queues also provide *location transparency*, eliminating the need for explicit remote communication, and *endpoint* transparency through the message router pattern [EIP:78], which enables more flexible addressing of receivers by logical names, message properties or even search criteria. By using a canonical message format [EIP:355], application-specific formats are translated to a common format before they are sent over the wire, thus abstracting from internal data formats and enabling collaboration between disparate, incompatible applications.

Messaging thus allows loose coupling of interested parties using a common communication infrastructure, and is used as the “backbone” in many tool integration solutions, such as [Karsai2003], which implements the OTIF standard. Modern message-based integration solutions provide a *message bus* where applications can be plugged in and communicate dynamically with other applications on the bus but also to common infrastructure services

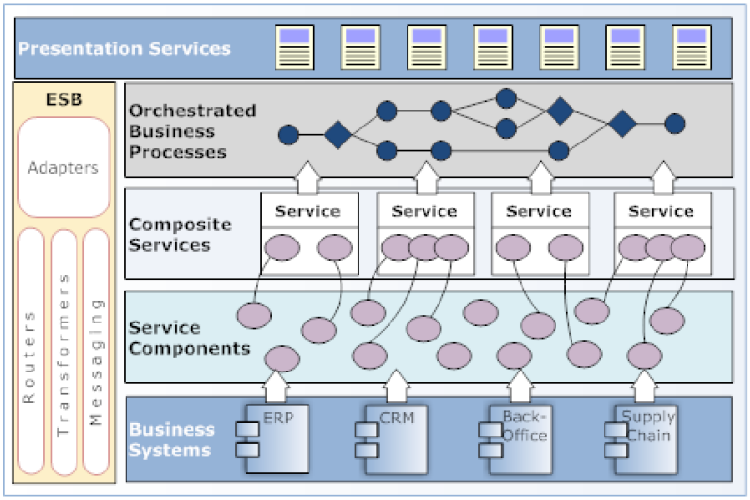
and frontends. A recent adaptation of this concept is the *Enterprise Service Bus* (see Section 3.3.3.2 below), a service oriented approach (see below) to enterprise integration facilitating a more abstract form of messaging.

3.3.3. Service Oriented Integration

As proprietary protocols, glue code, and point-to-point connections give way to more open, standards-based protocols and interaction based on service descriptions that each system externalizes, we step into the realm of Service-Oriented Integration (SOI).
--[Arsanjani2005], Toward a pattern language for Service-Oriented Architecture and Integration

Service-oriented integration is a novel concept that applies principles from service-oriented architectures to integration problems. In this context, integration is viewed as a "conversation between services"[39]. [EIP:8] defines a *Service* as "a well-defined function that is universally available and responds to requests from 'service consumers'". Abstracting from the rather narrow definition as a function, a more generic view is expressed in [Jones2005], who defines a Service as "a discrete domain of control that contains a collection of tasks to achieve related goals". [Erl2004] views Services as "independent building blocks", which in comparison to components are deliberately limited to implementing a single functionality that is provided for reuse by other Services.

Figure 3.8 provides an overview of a typical SOA environment, the individual concepts are explained in the following sections.



(from [Davis2009:11])
Figure 3.8: a service-oriented environment overview

[Arsanjani2005] introduces the basic concepts behind service-oriented integration and introduces several related patterns for integrating applications according to SOA principles. In this article, an ESB is simply an implementation of the SOI-pattern. Challenges of service-oriented integration, including problems with differences between local and remote communication, or coupling, are covered in [Trowbridge2004:146]. [Juric2007] provides a recent overview of enterprise integration concepts and patterns, in particular service-oriented integration, and gives several examples for applying web services and related technologies to real world integration problems.

By using established, service-oriented standards like *WSDL*, service-oriented integration enables advanced communication using interaction patterns, called *Message Exchange Patterns* (MEPs), which are applied in [Hoh-

[39] taken from the presentation slides Open ESB v2, Open ESB.next and Project Fuji [http://wiki.glassfish.java.net/attach/GlassFishDay2008Jazoon/OpenESBv2-Project%20Fuji.pdf]

pe2007] to realize rich conversations between services. MEPs are also proposed in Section 4.2.1 for realizing inter-tool-communication.

Alternatively to a web-services based implementation, a service-oriented integration can also be implemented by providing WSDL descriptions for existing (legacy) applications that are not web service-based, but connected through *Wrappers*. This approach is demonstrated by [Yap2005], using web-service wrappers for integrating client applications (with the example of jEdit).

Recent higher level integration-frameworks like WSIF, JBI or SCA (see below) follow this approach, which has been defined by Ron Ten-Hove[40] as “Service-based integration” that “works by modeling integrated applications as services.”. JBI consequently defines all service interfaces of both external and internal endpoints using WSDL. This form of integration provides a very powerful way to integrate legacy or closed applications such as COTS tools in a *loosely coupled* way, which makes this approach a compelling candidate for tool integration. Until now, these standards have not yet been used to realize a tool integration solution, but web services and WSIF have been successfully applied in large and small enterprise integration scenarios, see Section 3.3.3.1 below.

Looking at the Web, also “mash-ups” can be seen as an example of service-oriented integration, as existing services are connected in new ways beyond the creator's original intentions, such as the prime example of GoogleMaps and Flickr, connecting photos to geographic locations. The *OpenAjax Alliance* defines mashups as “a website or web application that uses content from more than one source to create a completely new service.” [OpenAjax]. A similar, user-oriented solution would be desirable for spontaneous tool integration that allows end users to combine tools as needed in a simple but powerful way. The following sections will shortly examine current solutions that may be useful for realizing web-like mashups on the desktop.

3.3.3.1. Web Services Integration

When applications are exposed as web services, it would be tempting to realize integration by simply accessing web services as needed. This naïve approach however has several drawbacks: services are tightly coupled, as they are directly connected; the resulting *point-to-point* integration does not scale well and is hard to adapt as services are replaced, upgraded or even relocated. Lastly, performance is an issue when integrating client side tools, and solely relying on web services would introduce a considerable overhead[41]. Consequently, [Erl2004] concludes: "Introducing Web services into an environment does not replace the need for middleware and many traditional integration technologies. Web services are not a new form of application integration or EAI, they simply add new components that can be utilized effectively in a variety of architectures.". [Vinoski2003] investigates integration using web services as a way to bridge incompatible legacy middleware solutions, but that often binds implementations directly to a specific SOAP stack (e.g. Axis, XFire or CXF). He then examines solutions above the protocol level, proposing a web service-based integration framework, the Web Services Invocation Framework (WSIF), as a high-level service-based integration solution.

Apache WSIF[ApacheWSIF] was proposed in [Duftler2001] as “an open source initiative to provide a service oriented framework that allows both SOAP and non-SOAP services to be described in WSDL and invoked in a common way. WSIF defines a pluggable interface […] to support new transports and protocols.”[42] WSIF has providers for POJOs (simple Java objects), EJBs, JMS message queues, JCA adaptors, and SOAP, but stays above the protocol layer, abstracting from different service-based APIs and from a web service-centric development view: “WSIF gives to its users a uniform API to access WSDL-described Web Services.” [Duftler2001], where “the only requirements are that the service be described in WSDL and that a relevant protocol binding implementation is plugged into the framework.”. By using WSDL as a common interface definition language,

[40] specification-lead for the Java Business Integration-standard, JSR 208 [JBI]

[41] see also Steve Vinoski's article Web services no interop cure-all [http://www.theserverside.net/tt/articles/showarticle.tss?id=WSNoInteropCure]

[42] from the article Applying the Web services invocation framework [http://www.ibm.com/developerworks/webservices/library/ws-appwsif.html] at the IBM DeveloperWorks site

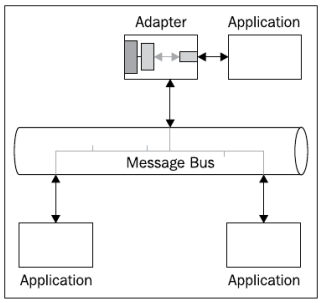
also for legacy endpoints, WSIF provides existing systems with a service-façade that allows transparent integration between web services and existing systems.

While WSIF provides a high-level API needed for building protocol-agnostic service clients, there is no matching server part, no messaging facility for providing advanced concepts like mediation or routing. This results in a static solution that cannot easily be extended, as services are more closely coupled, even when they are based on high-level interfaces that abstract from the underlying web service protocol.

WSIF, and web-service integration in general, therefore provide a client-side solution for small integration projects or for building more static, composite applications (see also Section 4.3.1), when the routing and mediation functionality of a full-fledged ESB (see Section 3.3.3.2) is not needed; for a tool integration framework however, this additional mediation (or middleware) layer is essential.

3.3.3.2. The Enterprise Service Bus

The concept of an *Enterprise Service Bus* (ESB) was introduced by David Chappell in 2003 with his landmark book [ESB]. An ESB provides a service-oriented integration infrastructure that applies SOA concepts throughout, as defined in [ESB:2]: "An ESB provides the implementation backbone for an SOA. That is, it provides a loosely ocupled, event-driven SOA with a highly distributed universe of named routing destinations across a multiprotocol message bus. Applications (and integration components) in the ESB are abstractly decoupled from each other, and connect together through the bus as logical endpoints that are exposed as event-driven services." Figure 3.9 provides a schematic overview of an ESB, illustrating how existing applications are integrated using Adapters:



from [Christudas2008:15]
Figure 3.9: Architectural view of an Enterprise Service Bus

Unlike pure web-service integration and related frameworks, services connected to an ESB do not directly call each other, but are more decoupled and communicate by sending requests and data messages over a common message bus. Services connect to an ESB by publishing their logical endpoint address, which is then made available by the ESB to all services connected to the bus. Also, message senders do not have to address a concrete target, but can rely on intelligent routing services on the ESB that direct the message to an appropriate endpoint. *Message routing* is an essential part of an ESB and many enterprise integration patterns are based on routing, which is explained in detail in [EIP:225]. On an ESB, communication may be synchronous or asynchronous, and services do not have to actively request information, as they receive services automatically when they are subscribed to endpoints of interest. This follows the event-driven consumer-pattern introduced in [EIP:498] and shows that an event-driven architecture can be realized side-by-side with a service-oriented architecture, using an ESB[43].

The message bus also provides functional integration and data integration by using concepts like *mediation*, where common infrastructure services or application-specific Adapters translate requests between incompatible

[43]see also the article Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus [http://www-128.ibm.com/developerworks/webservices/library/ws-soa-eda-esb/] at IBM developerWorks

protocols, and message translation, using a common, *canonical* message format that abstracts from incompatible data (the problem of incompatible data formats is often referred to as the “impedance mismatch”, c.f. [ESB:10]).

An ESB also enables dynamic service composition at a higher level, as shown in [ESB:2]: "Using an ESB, an integration architect pulls together applications and discrete integration components to create assemblies of services to form composite business processes, which in turn automate business functions in a real-time enterprise.". Composite applications are introduced in Section 4.3.1 as a core concept of the proposed solution. For a more detailed review of ESB characteristics and core functions, refer to [ESB:7] and also [Rademakers2008:12], which specifically addresses application integration with ESBs using JBI.

Because an Enterprise Service Bus implements many of the integration patterns described in [EIP], it is very well suited for a realizing *dynamic* tool integration, but all functionality comes at a cost: ESBs are often not easy to set up, configure and maintain, and a full-featured ESB may introduce too much resource overhead, especially when used for client integration. While a canonical message format provides many advantages, like decoupling data from the original application that created it and enabling message inspection, enrichment and transformation (c.f. [EIP:355]), the integration breaks when the canonical format evolves or has to be changed in such a way that it becomes incompatible to the old format. This is especially a problem when the canonical message format is proprietary, as this binds the solution to a specific *ESB* implementation (e.g., *Mule*, which is open source, but uses a proprietary component model and messaging format). This problem can be addressed by using open source ESB implementations that implement open standards, like Apache ServiceMix [ServiceMix] or OpenESB [OpenESB], which implement the JBI standard.

Even though ESBs are based on standards, there has been no standard for defining what an ESB is and how Services, Adapters connect to it, or how the message format and communication on the ESB should be implemented. This has led to various incompatible implementations with each providing their own set of proprietary Adapters, and their own canonical message format, etc. *Java Business Integration* (see Section 4.2) tries to solve this situation by specifying a common architecture that ESBs can implement in a standards-based way, enabling sharing of Adapters and other ESB components.

Figure 3.10 illustrates the relationship between service-oriented integration, JBI and ESBs, mapping the overlapping concepts introduced in this section as class interfaces in UML notation, and summarizing the key characteristics of each approach.

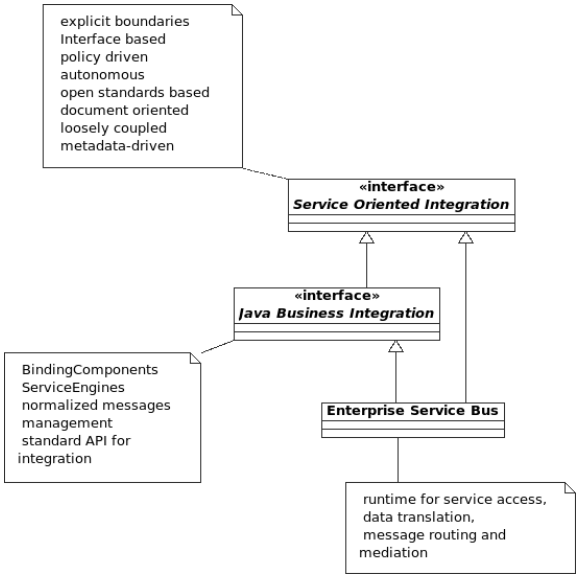


Figure 3.10: Relationship between SOI, JBI and the ESB

Lastly, [ApacheSynapse] could be an interesting alternative for implementing more static integration tasks analogous to pipes. *Synapse* is more flexible than, e.g., WSIF, and incorporates several ESB concepts such as mediation, translation and protocol abstraction, but does not provide advanced routing capabilities or dynamic composition like full-featured ESBs do. However, *Synapse* facilitates the design of lightweight integration solutions that are easier to implement.

Section 8.5 introduces an alternative approach building on REST and a new paradigm called “resource-oriented computing”.

3.3.3.3. Current Service-Oriented Integration Solutions

This section will give a short overview of existing enterprise integration solutions that show the current state-of-the-art in service-oriented integration in the enterprise.

While standards-based containers like JCA (see Section 3.3.7.1), plugin-frameworks like OSGi (see Section 3.2.3.1), or service-oriented frameworks like WSIF (see Section 3.3.3.1) or architectures like the ESB (Section 3.3.3.2) provide a rich foundation for service-oriented, modular integration that reaches out to legacy systems, they still require manual composition and glue code to form a working out-of-the-box solution that can be used for integrating various data sources and backend systems in an enterprise. There are several open source and also commercial solutions on the market that provide an integration and orchestration layer that allows transparently combining existing systems and data sources in a service-oriented manner, as shown below.

3.3.3.3.1. Open Source Solutions

[Apatar] (as introduced in Section 3.2.4) is an open source *ETL* (Extract, Transform and Load)-solution that provides a visual job designer and data mapper, connectivity to all major data sources and flexible deployment options (embedded, *GUI* or server)[44]. With Apatar, analysts can create data maps that define systems and sources to be integrated, and the associated workflow for gathering required data. These data maps can then be shared on an open community platform, where also existing data maps contributed by others can be imported for reuse in custom integration projects. [Jitterbit] offers similar data integration based on web services and also includes a graphical designer that is targeted at business analysts and allows the creation of integration pipelines.

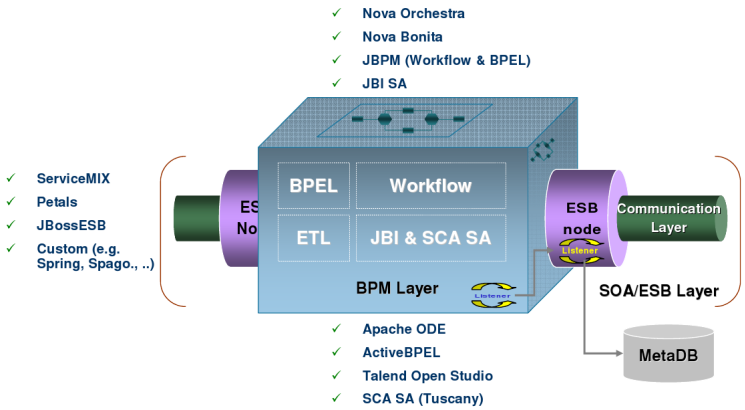
[XAware] is an XML based SOI integration solution that was initially closed but has become open source with version 5.[45] XAware provides an XML-based data integration layer, realizing “a heterogeneous data abstraction environment” (David Linthicum, ZapThink LLC). This environment can be used to create “data mashups” (Bill Miller, XAware). Disparate data from heterogeneous sources is translated into canonical XML data objects that are made available to other sources over a uniform, message-based data layer, and previously isolated data sources are exposed as services. Connectors provide protocol integration with HTTP, RMI, SOAP, or other protocols using the Java API. Adapters realize semantic data integration and provide transformation of commonly used formats such as flat text files, CSV, Excel, or COTS data sources like SAP. Integrated data is then made available for composition as logical views (that work like a meta-model) using a graphical designer based on Eclipse, where data and services can be composed to form the desired data integration solution. A proprietary scripting language (XA-Script) supports conditional logic, enabling work flow-integration.

[Spagic] takes a slightly different approach, maximizing reuse of open source integration solutions and providing a meta-platform for enterprise integration: “Spagic is a SOA Enterprise Integration Platform composed by a set of visual tools and back-end applications to design, develop and manage SOA/BPM solutions.” (from the Spagic web site). The open source EAI suite provides modules for designing business processes and for modeling and

[44] see the Apatar web page on Application Integration [http://apatar.com/for_application_integration.html]

[45] see also the article XML data integration for SOA goes open source [http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1280942,00.html?track=NL-130&ad=612051&asrc=EM_USC_2564354&uid=6341833]

dynamically generating composite services for realizing data and process integration solutions. Existing ESBs such as Apache ServiceMix, PEtALS, JBossESB; service components (adhering to the JBI or SCA standard), and data sources can be integrated, and Services can be orchestrated using a visual BPM designer based on Eclipse. Service modeling is supported by integrating the Eclipse Service Tools Platform (see Section 3.2.4.2). Also modules for *Business Activity Monitoring* (BAM) and process measurement are provided. The rather complex integration architecture is shown in Figure 3.11 below:



(source: [Spagic])
Figure 3.11: The Spagic open source enterprise integration platform

While these solutions provide flexible, service-oriented integration, they focus on the data and process level, leaving out higher-level integration necessary for general tool integration, such as *functional* integration (reaching through to the integrated tool) or *interface* integration (providing transparent extension of existing tools). The focus lies on translation of data and services, not in providing a common infrastructure for tool integration. The target domain is clearly backend integration, not user-centric integration. Spagic provides an interesting approach and integrates with many open standards and frameworks, but is very complex and may introduce too much overhead for the tool integration framework envisioned in Chapter 4, depending on how easily the framework can be modularized. Although high-level standards like JBI are officially supported, integration is only possible at deployment time, not at design time, as only the binding part of the JBI specification (see Section 4.2.1) is currently supported.

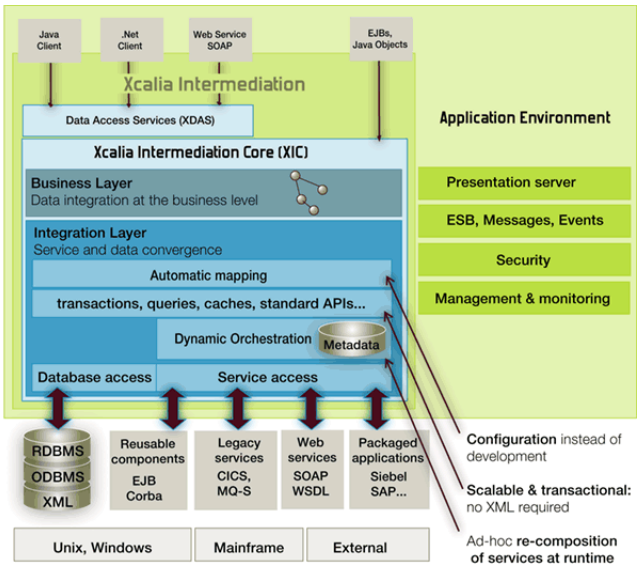
3.3.3.3.2. Commercial Solutions

IONA (now Progress) [FUSE] is an SOA suite that adopts several open source solutions, adds enterprise features such as advanced monitoring, management and commercial support, and provides an integrated suite including a dedicated enterprise integration designer based on Eclipse[46]. The package includes an ESB (based on Apache ServiceMix), a message router (ActiveMQ), a web service stack (Apache CXF) and an intelligent DSL-controlled router (Apache Camel) that implements established enterprise integration patterns. Using the graphical integration designer, Apache Camel integration rules can be designed visually and then deployed to the routing engine.

[Xcalia] provides a service oriented mediation layer for enterprise integration, focusing on integrating heterogeneous data, using JDO and the emerging SDO standard (see Section 8.2). The platform is built on top of an intermediation layer that implements recent standards like SDO for data integration and SCA for service-oriented integration and dynamic composition at runtime. Development of custom integration code is minimized by providing rich configuration possibilities, using a visual design-tool for object/relational data mapping and

[46] only available as a preview at the time of writing

a metadata based approach for object/service-mapping. Existing enterprise applications can be integrated using (proprietary and JCA) Adapters.



(source: [Xcalia])
Figure 3.12: Architecture of XCalia's service oriented integration layer

Xcalia supports both the Java and the .NET platform, and data integration can also be realized with the .NET query language LINQ (with VisualStudio integration). While Xcalia provides an interesting approach using open standards and dynamic service composition based on metadata-modeling, it does not provide visual support for service oriented integration and only operates on the data level, albeit on the logical business layer. Therefore, the same critique applies as for the open soure solutions above.

Lastly, looking at the .NET-world, solutions are usually built around Microsoft's BizTalk server that offers Adapters for SAP and other enterprise applications. Internal messaging is often handled using the standard *Windows Communication Foundation* (WCF) which is part of .NET, and external applications or Java-application(server)s are integrated using web services.

3.3.4. Workflow and Process Integration

This integration form is another example where enterprise and desktop integration overlap and has been introduced in Section 2.3.6. In the enterprise domain, process integration and the related discipline of *business process modeling* (BPM) has become a major driving force behind recent standards such as BPEL (now *WS-BPEL 2.0*) and related extensions like BPEL4People and *WS-HumanTask*, which provide service-oriented integration of non-automated, manual (or human) tasks, which have to be performed by humans (e.g., the acknowledgement of an insurance claim by an insurance clerk).

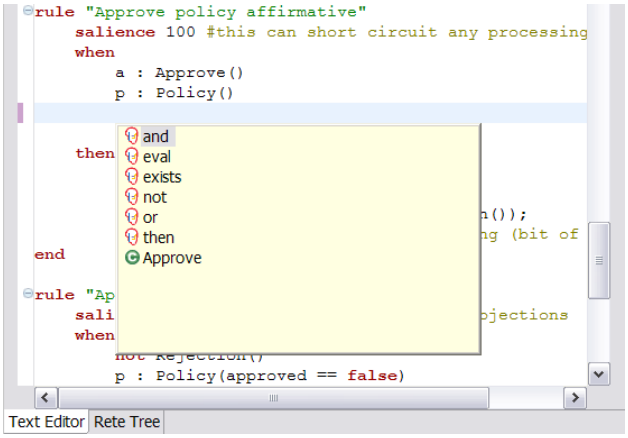
BPEL “defines a language for business process orchestration based on web services”[47] and is specified by the OASIS standard [WS-BPEL20]. The specification extends the static WSDL interface description that models a single service, describing its methods and properties, describing how individual services are combined to implement a business process.

Current (open source) implementations include [JBoss jBPM] or the BPEL engine [Apache ODE], which provides integration layers for embedding BPEL processes into different target domains, e.g., into a web services

[47] taken from the WS-BPEL Primer [http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.html]

environment (using the Apache Axis2), or into an ESB, e.g. the JBI-enabled Apache ServiceMix. Support for the SCA-standard is already in progress and will be implemented with support for Apache Tuscany (see Section 4.2.2.3).

Where a full-scale process definition like BPEL introduces too much complexity or there are other constraints (e.g. a complete process definition may not be available), *rules engines* (see Figure 3.13) provide a lightweight alternative both during development and also at deployment, as processing rules can be specified by using a simple rules-language that is executed by a rules engine (e.g., JBoss Drools [Drools] or Apache Camel[48]) for routing messages, e.g., on an enterprise service bus. Apache Camel additionally provides a mediation layer that can be configured using a Java-based DSL or Spring configuration, and that can be embedded into a web-service based environment (using Apache CXF framework) or into a JBI based ESB (with Apache ServiceMix), similar to Apache ODE mentioned earlier.



(source: [Drools], “Features and Screenshots”)
Figure 3.13: Workflow integration with rules-based programming

Process-oriented integration is supported by a wide array of tools, which are mostly based on Eclipse projects (e.g., BPM tools as part of the SOA Tools Platform, see Section 3.2.4.2.1), e.g., the *Eclipse Java Workflow Tooling* (JWT) Project, which integrates several BPM standards and notations. Spagic (see Section 3.3.3.3.1) is an open source enterprise integration solution that uses Eclipse STP for process integration. A related, model-based solution, the E2E bridge, is covered in Section 3.3.6 below.

[Raj2006] gives an introduction into using WS-BPEL in Java to realize service-oriented workflows defined by BPEL process descriptions. An extensive overview of process-oriented integration and service composition using BPEL is provided in [Juric2007:213].

To summarize, BPEL may be used for workflow-based tool integration where a concrete process has to be followed, and where tools interact according to well-defined rules. For realizing dynamic, *ad hoc* tool integration in a more generic way, where the end user has free control over the tools' usage, a fixed process definition is too static and limiting. For this, rules-based systems provide a flexible alternative that integrates with other integration standards and provides users with a dynamic way to specify and adjust workflows at runtime.

3.3.5. Event Driven Integration and SOA

In [Woolf2006], event-driven architecture (EDA) is defined as “a technique for integrating components and applications by sending and receiving event notifications.” An event is subsequently defined as “an occurrence

[48] see the Apache Camel page on routing [http://activemq.apache.org/camel/routes.html]

in one application or component that others may be interested in knowing about.". [Hohpe2006a] defines several key characteristics of events, including *broadcasting*, *timeliness* (as they happen), or *asynchrony*, and mentions that "these desirable benefits have already motivated some EAI […] vendors to proclaim that EDAs are the next step in the evolution beyond Service-oriented Architectures (SOAs).".

Analogous to service-oriented architectures, where a *consumer* sends a service-*request* to a *provider* that implements the desired Service functionality, in an EDA, an *emitter* posts an *event* that is received by a *handler*, which decides how to react upon it and which, if any, service(s) should be invoked as a consequence. While there is a direct relation between the service consumer and the provider in an SOA, following a *request/response* interaction style, there is no direct mapping from an event *emitter* to an event *handler*: although an event emitter must be connected to at least one event handler in order to be able to submit events, connections are mostly indirect, in a *publish/subscribe* manner. As such, event handlers subscribe to events of interest, but the emitter does not address a specific event handler. This allows more loose coupling than with a traditional SOA approach. JMS message topics and queues are an example for an implementation of an event-driven infrastructure using message queues for transmitting event messages.

SOA and EDA are both working with services but handle communication differently (service request chains vs. event propagation). Both models are however more complimentary than competing, as noticed in [Woolf2006]: "[…] for a sufficiently complex integration solution, one might well use both architectures.". Advanced event-driven concepts include *complex event processing* (CEP), enabling analysis of *event clouds*, where a multitude of events is distributed across multiple systems, and event stream processing (ESP), providing correlation of an infinite set of events that happen in realtime.

[Esper][49] provides a robust, open source implementation of a complex event processor. [Welsh2002] proposes a staged event-driven architecture (SEDA), which enables the processing of massive amounts of events in a short period of time, e.g., spikes in web site traffic, using a network of event-driven *stages* that are connected to individual, structured *event queues* for processing incoming events.

Event-driven systems also may impose challenges regarding design and complexity, as noted in [Hohpe2006a]: As event messages are distributed across many nodes and components, it is increasingly hard to analyze or predict the flow of execution, which makes it hard to find the cause for problems, especially when configuration is spread across disparate locations. The work suggests using a DSL for configuration, and underlines the need for design tools that validates a composite event processing system for unwanted configurations (e.g., cyclic event paths).

EDA is very useful for tool integration when the emphasis lies on general tool services in loosely coupled, dynamic tool chains, where tools may be replaced, come online or go offline at runtime. [Liu2006] developes visual languages and design tools for event-based tool integration using web service composition and data integration with *Abstract Data Structures* (ADS).

3.3.6. Model Driven Integration

As introduced in Chapter 2, *model-driven* integration offers promising potential and has been subject to intensive research during the last years, with first commercial products already available on the market (see below). Because many tool integration solutions, including part of the proposed solution, are based on model-driven concepts, esp. the process-based design pattern introduced in [Karsai2003] below, this integration technique will be covered in more detail here.

Modeling a COTS integration solution requires a thorough understanding of the application model, which is not easy to gain, especially from closed COTS tools where documentation on the internal structure and source code is usually not available, which makes it impossible to automatically generate needed modules, e.g., through reverse

[49] see also the related introductory article [http://www.onjava.com/pub/a/onjava/2007/03/07/esper-event-stream-processing-and-correlation.html]

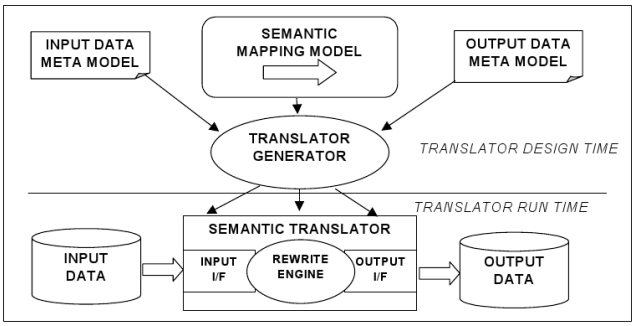
engineering. This is a major drawback since the automatic mapping between model and implementation is one of the main reasons to use model-driven concepts in the first place. [Warboys2005] proposes a framework for integrating COTS tools, introducing a dynamic modeling approach that uses an architecture modeling language for handling constantly evolving tools.

[Balasubramanian2006] applies a model driven approach to tool integration by developing a domain-specific modeling language called SIML, which is then applied to enterprise integration, building on the open-source *Generic Modeling environment* (the GME) for visually designing the integration solution. As an advantage, no programming is necessary, even "glue" code required for integration is created automatically; also integration with BPEL and other orchestration standards (see Section 3.3.4) is possible. The prototype integrates with web services and the now largely obsolete CORBA *component model* (CCM), but offers no integration with COTS tools, since the approach relies on the source code to be available. While the work acknowledges this aspect and also mentions emerging standards such as *Java Business Integration* or the *Service Component Architecture* (described in more detail in Chapter 4) as emerging "pluggable architectures for system integration", these findings are not applied to the proposed solution.

The effort and cost involved with modeling could be reduced if existing models were reused, as proposed in [Denno2003] who observes that "traditional integration makes little or no use of the models, which were created at great expense and which provide valuable information about a system.". In the same way, [Mellor2003] foresees that a "software development environment with off-the-shelf models and mapping functions changes the way in which we build systems."

Related standards have been long missing from modeling approaches. This has changed with the introduction and widespread adoption of UML, the *OMG* standard for models in software development. The same organization has also proposed a standard for modeling tool integration: The Open Tool Integration Framework [OMG2004] defines a framework and architecture for model-driven integration through integrating metamodels of individual tools and using a shared repository for common data exchange. The standard "seeks to define an alternative to the closed tool suite approach, via an open tool integration framework that provides a platform for integrating a wide variety of tools, is open and extensible, and supplies generic, reusable facilities for building tool integration solutions." (from the OTIF RFP [OMG2004])

[Karsai2003] demonstrates how meta-model transformation could be applied to tool integration, introducing two design patterns for tool integration: *Integrated Data Models* (IDM) and Integration based on *process flows* (see also Section 3.3.4). The first pattern is illustrated in Figure 3.14 below:

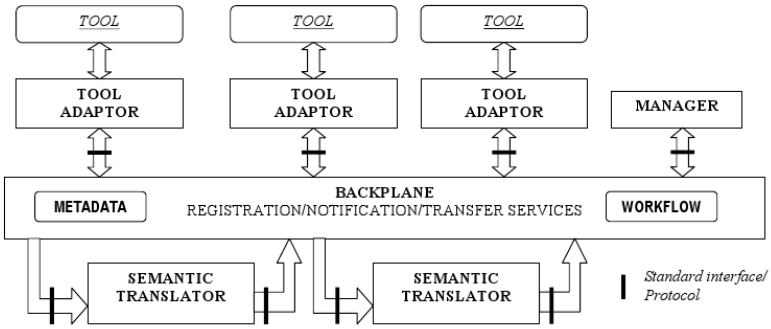


(taken from [Karsai2003])
Figure 3.14: Model driven integration using metamodel transformation

Many model-driven tool integration approaches follow this pattern. While it is useful for tools with overlapping data models that share common semantics, e.g., tools from the same domain, this pattern introduces a major problem: as tool integration solutions grow over time, the number of integrated tools increases, and the resulting

solutions do not scale well. Consequently, the work concludes that “practical experience with the IDM approach showed that it becomes very complicated if the number of tools grows beyond three or four. To understand and maintain the mapping […] is becoming an insurmountable task for an engineer.”.

The second design pattern, *process-based* integration, follows a “point-to-point” approach (but using a message bus) and integrates individual tool models using Adapters. Data is shared using *Semantic Translators* connected to a message-based backbone, as shown in Figure 3.15:



(taken from [Karsai2003])
Figure 3.15: Process-based tool integration using a common backbone

This approach “does not have these shortcomings, as the changes are always localized. Changing a metamodel for a tool impacts only the translators that read and write models of that tool, but not others.”. The process-based approach is therefore more suitable for general tool integration, as it is more loosely-coupled and does not assume any semantic relationship between tools. Also the aforementioned OTIF framework is following this approach. [Klar2008] strives to improve the design of model-based solutions and introduces a more formal process for designing metamodel-driven tool integration solutions, also considering COTS tools.

A commercial, model-based enterprise integration solution is available with the E2E Bridge (for End-to-End), a middleware solution that acts as a virtual machine for models specified in UML[50][Baer2007]. E2E proposes a purely model-based approach for designing enterprise integration solutions that can be automatically deployed without manual implementation or code transformation. The concept is described as “Direct Model Execution based on standard UML, BPMN, EPC and other modeling languages […] used […] to implement and manage distributed software assets in support of automated business processes.”. Unlike other approaches which transform models into code that is then compiled or run in a VM, E2E transforms the model into an exectutable form that runs in its own VM, therefore eliminating the intermediate code-generation step which often introduces inconsistencies between the model and the generated code. This results in a much more dynamic and decoupled solution that realizes the full potential of UML.

It is important to keep in mind that model-based integration alone is not enough for a holistic tool integration approach that transparently exposes not only the individual tools' *data*, but also makes available the combined *functionality* provided by integrated tools to users. Model integration really only solves the *data* integration problem, but does not provide other kinds of integration such as on the functional, process or the interface level. For this, still custom *Adapters* and other techniques are needed, as again noted by [Karsai2003]: “The primary motivation […] is […] to facilitate tool data interchange.”.

Also, where existing Adapters are already available, they cannot be used without first constructing metamodels for integrating them into a model-based solution. Especially in an SOA landscape, where functionality is already

[50]see the E2E-Bridge web site [http://www.e2ebridge.com/]

exposed through web services, there is no need to re-model these integration points since abstract models are already provided by WSDL definitions, which can be readily used with modern integration frameworks such as SCA and JBI.

Another challenge is tool support: modeling environments have typically been sophisticated high-level tools that are either commercial and costly, or academic and not available for use in production. Existing modeling tool sets often use proprietary formats and only provide restricted interfaces, which results in isolation of model data and workflows.The *Model Driven Development integration* (MDDi) project was introduced to fill this gap, as it "produces an extensible framework and exemplary tools dedicated to integration of modeling tools in Eclipse"[51]. The project was created as part of the EU ModelWare project, "an open source tool integration platform that facilitates the customization of MDD tool chains for domain-specific needs".[52] Resulting solutions include a Java-based QVT implementation that is now part of the Eclipse EMF project, and the *ModelBus* [Sriplakich2008], which integrates heterogeneous modeling tools inside (using XMI) and outside of Eclipse (using web services and Adapters), and supports distributed collaboration on models.

These projects could be used as a starting point to build a specialized environment for COTS integration, as suggested in [Kramler2006] and applied for embedded systems in [NascimentoS2007], using the EMF and related projects such as MDDi as foundation (see also Section 3.2.4.2.1 above).

Lastly, Sculptor[53] provides a design environment and iterative code-generation for model-driven application development, building on the Eclipse-based openArchitectureWare project (the see aforementioned section).

3.3.7. Standards-Based Integration

This section gives a short overview of two well known standards approaches in enterprise integration: the component-oriented JCA standard predominant in the JEE world, and second generation web-service standards for service-oriented integration.

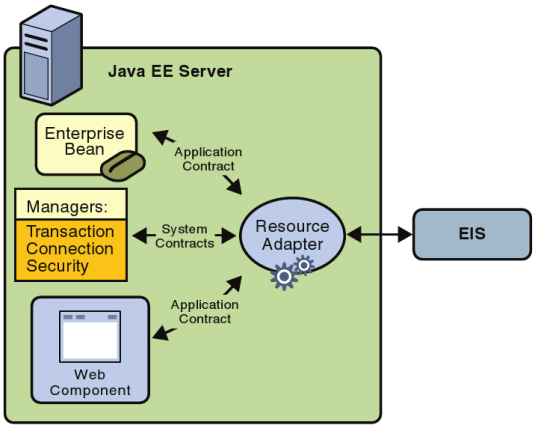
Higher-level integration frameworks that build on several standards and concepts introduced in this chapter will be covered in more detail in Chapter 4, including *Java Business Integration* (JBI), which focuses on integrating existing components in a runtime environment, and the *Service Component Architecture* (SCA), which defines a standard for service composition.

3.3.7.1. Java Connector Architecture (JCA)

> *The Connector architecture enables Java EE components to interact with enterprise information systems (EISs) and EISs to interact with Java EE components.*
> *--The Java EE 5 Tutorial*

The *Java Connector Architecture* [JCA] provides integration of external resources into a JEE-environment by providing a standard contract and API to connect *legacy* applications to application servers that support JCA (e.g., JBoss, Apache Geronimo, Sun Glassfish or IBM WebSphere), using a proprietary (application-specific) resource-adapter. Existing backend systems (like databases, but also proprietary solutions such as telephony systems or printing facilities) are connected to the application server using a standard component, the JCA *Resource Adapter*, that implements a set of contracts specified in the JCA specification. These contracts define various aspects important in the enterprise environment, such as security, management (lifecycle, threading) and inbound/outbound transaction contracts that define the communication with the external system to be integrated, as illustrated in Figure 3.16 below:

[51] taken from the MDDi Wiki [http://wiki.eclipse.org/Mddi], which is now archived as the project has been terminated as of August 15th, 2008
[52] from the ModelWare web site [http://www.modelware-ist.org/]
[53] see the Sculptor Wiki [http://fornax-platform.org/cp/display/fornax/Sculptor+(CSC)]



(source: [JEE5Tut:1020])
Figure 3.16: JCA Resource Adapter design

The *Resource Adapter* provides legacy integration with proprietary third party interfaces, but is embedded into a standardized Container, which allows reuse of Resource Adapters across application servers. [ESB:187] provides a helpful analogy to another JEE middleware standard: “JCA is to applications what JDBC is to database connectivity.”

When connected to an ESB (see Section 3.3.3.2), JCA “can also provide the standard contract for application adapters”, acting as “the unified way of connecting between the adapter and the middleware infrastructure” [ESB:54]. More details on JCA can be found in [JEE5Tut] and in the specification [JCA]. While JCA is widely used for integrating existing non-Java applications into JEE solutions, JBI's `BindingComponents` provide a more flexible way that further reduces coupling in such integration scenarios, as shown in Section 4.2.2.2.

3.3.7.2. WS-I and WS-*

The *Web Services Interoperability Organization* (*WS-I*) [WSI] is an open industry organization working on standards to ensure web services interoperability between different platforms and implementations. While these specifications standardize the *protocol* layer (SOAP, WSDL, UDDI), e.g., through the Basic-Profile specification[54], web service-standards are specified by the W3C, which also defined the second generation WS-* specifications such as WS-Addressing, WS-Security or WS-Messaging. [ESB:230] gives an overview of ESB features and corresponding WS-* specifications that are explained in more detail in [Erl2004:90], where they are defined as a “combination of next-generation Web-services to allow for secure, reliable and collaborative service interaction as needed in an enterprise environment. In this context, the term Service Oriented Enterprise is often used”. A *service-oriented enterprise* is defined in [Erl2004:476] as thus:“Building and integrating SOAs leads to the evolution of a service-oriented enterprise (SOE).”

The WS-* specifications were necessary to address several requirements that became apparent when web services were increasingly used for enterprise integration, such as security, reliability or more abstract addressing schemes, in a service-oriented manner. Due to their complexity however, they have not yet been widely adopted, and alternative, more lightweight approaches like REST (c.f. [Fielding2000]) and the Web Oriented Architecture are currently gaining popularity, see Section 8.5 for a short outlook on this new paradigm.

For implementing web-service integration solutions, these specifications provide advanced and often needed higher-level communication mechanisms. Open-source implementations of WS-* standards are available from the Apache Software Foundation's Web Services project[55].

[54] see the WS-I Basic Profile pages [http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile]
[55] http://ws.apache.org/

3.4. Summary

The current situation shows a trend towards embracing open standards at every level of integration, and from previously isolated and tightly coupled integration approaches to higher-level integration frameworks that combine several lower-level solutions and patterns. Also, visual integration tools are starting to emerge, also in the open-source world, mainly due to efforts on the Eclipse platform, such as the Eclipse SOA Tools Platform for designing composite service solutions and Project Swordfish as a universal SOA runtime platform. The highest-level of integration is promised by model-based solutions, but the potential has not yet been realized on a general level, as several attempts to deliver integration solutions have failed in the past, like the Eclipse MDDi project. In specialized business integration markets however, successful solutions have been deployed, which is demonstrated by the E2E productline.

For a general, open tool integration framework that integrates up to the user interface level, current solutions are not yet satisfactory, as has been shown for Eclipse-based and other open source solutions. Hence the need for a solution that fully embraces recently emerging high-level integration frameworks such as JBI, which provide a standardized infrastructure and build on proven service-oriented integration standards and patterns, as will be shown in the following chapter.

Chapter 4. Proposed Solution: Tool Integration Using Java Business Integration

"Always design a thing by considering it in its next larger context – a chair in a room, a room in a house, a house in an environment, an environment in a city plan."
--Eliel Saarinen, Finnish architect and city planner

Building on successful integration solutions and best practices presented in Section 3.3 on the one hand, and on modern, service-oriented concepts (as introduced in Section 3.3.3) on the other hand, recent years have seen the emergence of high level integration frameworks, the main proponents being *Java Business Integration* (JBI, JSR-208) and more recently, the *Service Component Architecture* (SCA). These frameworks specify a language- (esp. SCA) and platform-neutral (esp. JBI) integration infrastructure that utilizes service oriented standards and integration concepts to provide a truly open integration platform that can be easily adapted to individual needs. This frees integration developers from having to manually hard-wire services and applications together, using custom "glue" code, and instead allows to rapidly build standards-based, loosely coupled integration solutions.

This chapter investigates the design rationale and requirements for successful tool integration as outlined in Chapter 2, and provides a survey on the two major integration standards, JBI and SCA, together with an evaluation against the requirements defined. Finally, for reasons outlined below, the JBI standard is proposed as a solution, which is then applied in designing the tool integration prototype presented in Chapter 6.

4.1. Requirements

"The essential components of an IDE are the tools which have to be integrated. Any other component must serve for integration purposes."
--from [Altheide2002]

A general guideline for analysis of service-oriented legacy integration is given in [Erl2004:346], which may act as a starting point for requirements-gathering. [Young2003] provides an in-depth reference for requirements engineering in general, defining several types of requirements, roles and processes for successful requirements design.

The remainder of this section lists several functional and non-functional requirements for tool integration, going from general requirements to special requirements important for client-side tool integration and COTS tools. Where applicable, analogies to ToolNet (see Chapter 5) are drawn, and findings from the proposed solution are applied. Section 6.2 outlines specific goals for the prototype, which were identified in evaluating ToolNet.

When investigating these requirements, it is important to keep in mind that there are two perspectives in tool integration, as identified by [Thomas1992]: *tool users* ("environment users") and *tool integrators* ("environment builders"). As a result, not only have the requirements to be weighted against each other in the design phase, but also the user's and integrator's situation: while the first is looking for transparency in functions and data and seamless workflows crossing tool borders, the latter is interested in flexible tool APIs and a framework that facilitates integration and reuse, minimizing effort and cost.[Thomas1992] defines criteria for good integration at each of the layers introduced in Section 2.3: presentation, data, control and process. Special attention is also given to *usability*, which can be taken as a prime requirement for tool integration that affects both users and developers of an integration framework and has a major impact on productivity and effectiveness of the solution.

According to [Trowbridge2004:2], “an enterprise's integration architecture balances the requirements of the business and the requirements of individual applications”, concluding that “[…] the ideal [integrated] application is a thin layer of presentation that consumes shared functionality or data at the enterprise level.”. This means that existing functionality should be reused as much as possible, and new functions should again be exposed to other applications for further reuse. Because the same goals also apply to desktop tool integration, as has been shown in Section 2.3, they are adopted in the proposed solution here.

[Brown1992] defines several key requirements for tool integration, which are applied to the context of this work below:

- *Generality:* The solution should not be tailored to a specific domain or tool set, but allow for broad tool support. Although ToolNet's target domain is clearly technical engineering, this requirement was not mandatory for the proposed solution, nevertheless a general applicability was aimed for.
- *Flexibility:* The solution should be flexible to support a wide range of users; this was only of minor importance for the prototype and also for ToolNet, as users are mostly engineers. Nevertheless, care was taken so as not to limit the proposed solution in its flexibility.
- *Homogeneity:* The solution should provide users with a uniform interface to different tools and services. This can be realized by adhering to a consistent way of realizing *presentation integration*, e.g., by using common terminology, symbols, layout, behaviour, and functions. Also by providing a central management and query interface like the ToolNet Desktop, the user can interact with integrated Tools in a uniform way. The JMX console implemented in the prototype (see Section 6.4.2.4) reaches even further by adhering to the JEE management standard, which provides transparent integration within an existing management infrastructure.
- *Portability:* The solution has to work on different platforms; this is also a prerequisite for generality, as some Tools are only available for a particular platform or OS. For this reason, ToolNet and also the prototype are realized in Java.
- *Compatibility:* The solution should be able to adapt to existing tools and allow for soft migration. This is most important for COTS integration and is also identified in [Altheide2002] below.

[Altheide2002] adds the following requirements, which were the primary guideline for the realization of the prototype in Chapter 6:

- *Capability:* The solution should support core integration tasks across tools in an automated way, such as data exchange and query, maintaining consistency of data, and support for process integration
- *Flexibility* (or *Adaptability*, to differentiate from the related requirement of flexibility in [Brown1992] above): The solution should adapt to the dynamic tool market and allow upgrading and exchanging tools as needed. Also, it is not uncommon to have multiple versions of a tool running in parallel, e.g., during migration.
- *Extensibility:* The solution should allow incremental adjustment to individual development needs (e.g., tool sets), methodologies and processes.
- *Modularity:* The solution should represent “no monolithic integration of tools but a framework of cooperating components” [Altheide2002:3], which is different from IDEs that generally aim at providing a meta-tool composed of individual tools, forming a tool chain or tool “cloud”. By integrating tools as autonomous components, the integration solution is decoupled from individual tool's needs. This makes it possible to integrate COTS tools without compromising the flexibility of the integration framework and impeding the realization of other requirements.
- *COTS Integration:* this requirement correlates with [Brown1992] and underlines the need to integrate existing standard tools which cannot be easily replaced with “integration-friendly” or custom developed, open tools.

- *Rich data integration:* Because tool integration makes no sense without *data integration*, this aspect should be realized in a rich and powerful way: Users should not only be able to access data from all integrated tools, but also define relations between data elements, thereby creating a (n:m-)mapping from data elements in one tool to corresponding data elements in another tool. Data should be kept inside the original tools, and only references and other metadata should be stored in a common repository. Users could then query the repository for data and navigate relations across tool boundaries, using integration facilities within the original tools (e.g., scripts or other extensions). Data integration must work even if a tool is temporarily unavailable, or if a tool becomes available during a work session.

- *Presentation integration:* Integration functionality must be integrated into the original tool's user-interface, so that the user perceives a transparent workflow without leaving the original tool.

The last three requirements, COTS integration, rich data integration and presentation integration, are one of the main features of ToolNet that set it apart from other approaches so far, and the proposed solution seeks to stay true to this vision. Finally, [Altheide2002] also mentions that an integration solution should be generated in a model-driven manner, using UML diagrams for data modeling and UML's *Object Constraint Language* (OCL) for expressing relational constraints. This approach is still subject of ongoing research, as discussed in Section 3.3.6, and is neither applied in ToolNet nor in the prototype.

Not explicitly mentioned but nevertheless important is *transparency: end users* should not perceive a notable overhead in using the tool integration solution, they should be able to use their original tools as usual, without having to use tools in a special way or launch them from within the integration framework. Users should be shielded from the framework itself and the underlying communication backend, so they should not have to know explicit endpoint addresses or have to differentiate between using local or remote tools. *Developers* should not be concerned with lower levels of the framework, e.g., what implementation of a message queue is used, or how the message format exactly is structured. They should not have to actively manage state, e.g., polling for changes in other tool's data or user input. Instead, a high-level API should provide the necessary abstractions and logical concepts to ensure a consistent, reusable and efficient integration solution.

Because of the many similarities between tool integration and enterprise integration, we can also apply the findings in [EIP:39-41], resulting in the following requirements:

- *Loose coupling:* “Integrated application should minimize their dependencies on each other so that each can evolve without causing problems to the others.”. This is a prerequisite for the requirement of flexibility in [Altheide2002] above.

- *Balanced intrusiveness:* A good tool integration solution has to find a compromise between the (perceived) degree of integration, and the amount of effort needed to realize the desired integration. Changes to applications that are to be integrated should be minimized, for COTS applications this is usually not an option at all. High intrusiveness results in integration overhead, which impedes flexibility and extensibility, whereas a low intrusiveness results in insufficient integration, resulting in a poor user experience and lack of presentation integration.

- *Technology selection:* Designing and implementing a tool integration solution involves several decisions that affect how well the other requirements can be fulfilled. Choosing closed or proprietary solutions hinders portability, generality, extensibility and compatibility. Solely relying on custom solutions may have the same effect in the end. Many problems with ToolNet result from the use of proprietary or closed solutions, and implementing a custom architecture where now open standards and APIs exist (see Section 5.5).

    - For successful technology selection, a closely related criteria is *support*, both in terms of tool support and vendor (or community) support, but also available implementations (preferably a variety of solutions that implement a common standard). By using a supported and visible technology, integration *Adapters* and

Services may already be available for reuse, and realizing new solutions becomes easier with adequate tool support that builds on a proven, reliable development infrastructure.

- *Common data format:* For realizing data integration, it is essential to work with a common data format at the integration layer. This is both a challenge and a necessity, given the number of mostly proprietary and incompatible data formats used in COTS tools. By using a normalized data format at the integration layer, and *Adapters* that convert between the tool's proprietary data format and the normalized format, this challenge can be overcome. The proposed solution uses a standardized (*JSR-208*) and normalized (XML) message format to avoid being bound to a custom format that is not supported elsewhere and may prove too limiting over time.

- *Data timeliness:* Integrated applications should share data in a timely manner in order to minimize the danger of data getting out of sync. This requirement also ensures usability and transparency, as users do not perceive latency in accessing tools, which ensures a smooth and responsive workflow.

- *Functional integration:* As shown in Section 2.3.3, data integration is not enough to provide the desired tool integration. Also, the tools' functionality has to be exposed over common framework services, allowing users to access other tools' functionality from within the tool at hand.

- *Asynchronous communication:* For providing an uninterrupted and transparent user experience, but also for improving inter-tool-communication, esp. with remote communication, asynchronous communication should be provided, e.g. using messaging or an event-driven architecture.

- *Reliability:* Closely related to remote and asynchronous communication, a tool integration framework should ensure reliable communication, even when a tool is temporarily unavailable. This can be implemented using reliable message queues, as applied in the solution below.

Finally, additional requirements for integration can be applied from [Sun2004:16]:

- *Openness and Code Portability:* relates to the requirement of portability above, but emphasizes on openness of software development tools and artifacts, using open standards

- *Scalability* has not been explicitly mentioned above but is a result of modularity, loose coupling, asynchronous communication and using a common data format. Scalability is a prerequisite for flexibility and extensibility.

- *Business Agility* means that an integration solution should be highly configurable without having to change the implementation of components, using high-level configuration and tooling.

Security is another important aspect when dealing with sensitive data over an open network, e.g., when implementing distributed tool integration across several locations. For the prototype implementation, security was not considered in order to keep complexity low and to facilitate debugging. The selected technology however supports several aspects of security, as mentioned in Section 4.3.2 below.

With these diverse requirements in mind, and based on the review in Chapter 3, the JBI standard was found to provide an excellent foundation for a tool integration framework that allows dynamic and collaboration among COTS tools within a heterogeneous system landscape, as described in the following section.

4.2. An Introduction to Java Business Integration

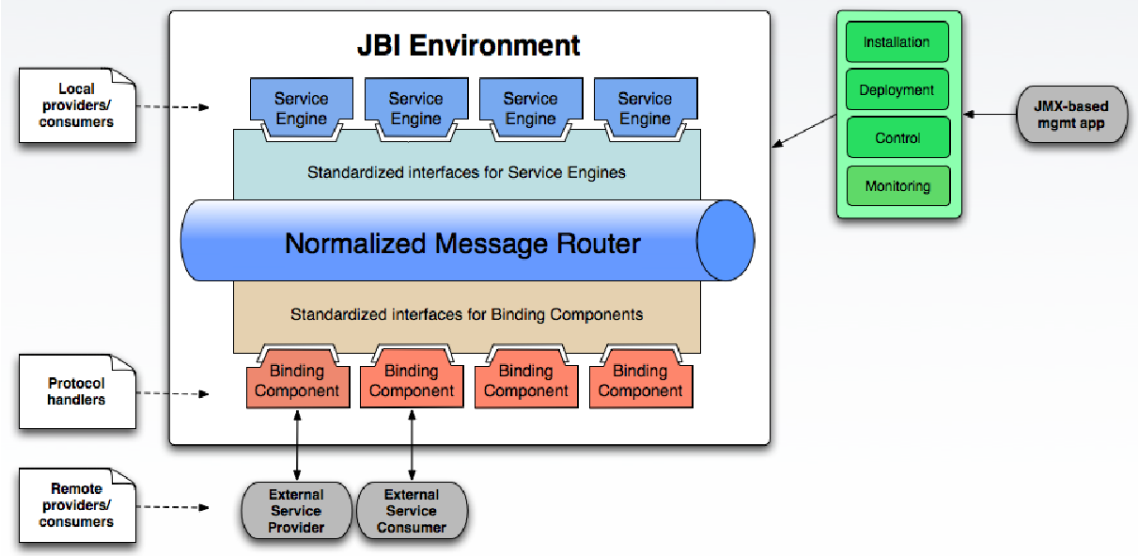
JBI defines an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange.
--The JBI 1.0 specification [JBI]

Java Business Integration [JBI] is a relatively young specification[1] by Sun, Inc., that specifies a standard API for a service-oriented integration-architecture and associated infrastructure, in order to harmonize the fragmented enterprise integration space and to avoid vendor lock-in that users are facing with current, proprietary ESB- and SOI-solutions (see Section 3.2.4.3 and Section 3.3.3.3.2). JBI defines a standards-based integration layer where components can be plugged in and work together seamlessly in a servie-oriented manner, exposing existing, external applications or services as *Service Endpoints*. Also legacy applications can be integrated in a service-oriented manner by providing a standardized service description (WSDL) and an associated integration Adapter (see Section 4.2.1 below).

JBI allows integration architects and developers to build integration solutions by combining existing components into *composite applications* without the need for implementing “glue” code or for manual “plumbing”. Instead, by relying on the JBI architecture and its messaging infrastructure, components can transparently communicate and invoke available services without knowing any protocol or implementation details of the target endpoint. Because JBI can be seen from the outside as an integration container that again hosts a set of specific integration components that may again be containers (such as EJB containers or JCA Adapters), JBI is also called a “meta container”.

As a result, JBI provides a standards-based solution for an *integration middleware* that overcomes problems in existing integration solutions, such as proprietary ESBs, “by adopting a service-oriented architecture (SOA), which maximizes the decoupling between components, and creates well-defined interoperation semantics founded on standards-based messaging.” (from [JBI:1]).

4.2.1. JBI Architecture



(source: [Snyder2007])
Figure 4.1: Java Business Integration Architectural Overview

The JBI architecture defines an infrastructure for service components connected to a message router that transports normalized XML-based messages. Developers build integrated applications (called “composite applications” because they are composed of existing services) by creating a configuration that references needed service components and includes necessary configuration and artifacts for the target component(s). Figure 4.1 gives a high-level overview of the JBI architecture (which is also explained in [Christudas2008:39]).

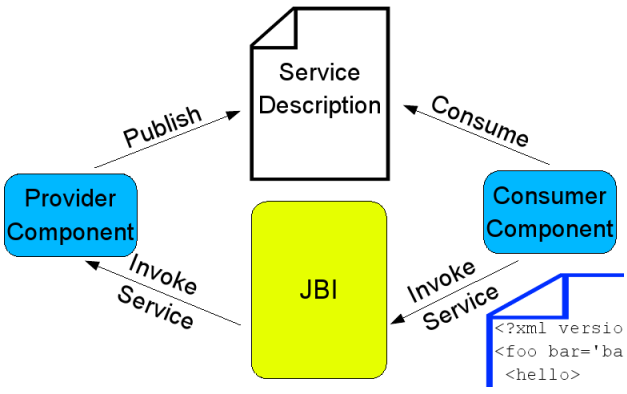
[1] the specification was finalized in August 2005, but only recently vendor and tool support has gained momentum

JBI defines two kinds of components: *BindingComponents* (BCs) integrate external services into the JBI environment by translating between application-specific communication necessary to access the external resource, and JBI's normalized messages (see below) in both directions, realizing protocol integration. This allows integrating and reusing existing assets, e.g., web services by using a SOAP BC, Enterprise Java Beans using a JavaEE BC[2], or proprietary applications like SAP using the SAP BC[3] (this has been utilized in the solution for realizing tool Adapters, as shown in Section 4.3 below). The second kind of component is the *ServiceEngine* (SE), which provides *application-level* and *process-level* integration and implements application logic, like conversion (e.g., an XSLT SE), orchestration (e.g., a BPEL SE) or event processing (e.g., there is an IEP SE, which implements an intelligent event processor). ServiceEngines do not perform any translation at the communication level, they always operate on normalized messages used within the JBI environment, and its XML (meta)data.

JBI is based on the concept of *service oriented integration* (as introduced in Section 3.3.3). As a result, service definitions are entirely based on WSDL. Ron Ten-Hove, the specification lead of JSR-208, motivates the service-oriented design approach and usage of WSDL as thus:

> Service-based integration works by modeling integrated applications as services. A WSDL declaration of a service provides all the information about that service to the service's consumers, such as the list of available operations, message formats, and so forth. Limiting a client's knowledge of a service to that service's WSDL definition was a very deliberate choice in JBI's design.
>
> —from [Sommers2005], section *JBI versus traditional system integration approaches*

`ServiceEngines` and `BindingComponents` publish the services they provide through a WSDL. When a component wants to invoke a service (as a *Service Consumer*), it can do so by specifying the logical endpoint name, a Service *Type*, or dynamically through a logical call-back address (c.f. [JBI:26-27]). JBI's message router (see below) handles discovery of the Service Provider and delivers the service request to the target endpoint. This is similar to the *find–bind–invoke* paradigm in a SOA, illustrated below:



(source: [Ten-Hove2006])
Figure 4.2: JBI's service-based integration model relying on SOA principles

Using a widely adopted and proven service-oriented standard such as WSDL for defining component interfaces and message formats has several advantages, such as loose coupling between integrated applications and Services, a common contract for enabling message exchange, and message mediation by utilizing WSDL's message exchange patterns (c.f. [JBI:11]). Because JBI fully embraces the WSDL standard for defining Service interfaces and their interaction, it also follows the distinction of an abstract and a concrete part. Figure 4.3 visualizes the relation between abstract and concrete definitions in a WSDL:

[2] see OpenESB Wiki's JBI4EJB page [http://wiki.open-esb.java.net/Wiki.jsp?page=EJBBC]
[3] see OpenESB WIki's SAP BC page [https://open-esb.dev.java.net/SAPBC.html]

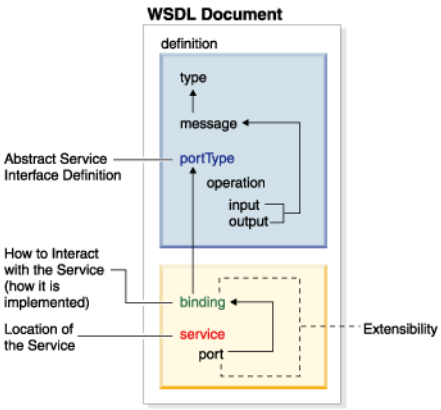
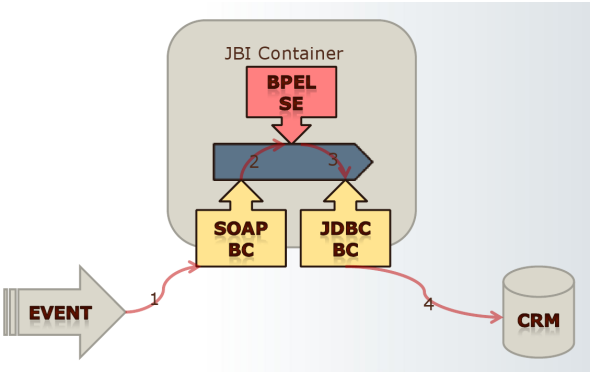


Illustration of the mapping between abstract and concrete part in a WSDL (source: Apache WSIF-project [WSIF])
Figure 4.3: Using WSDL for service-oriented integration

The abstract part is used for describing the protocol-neutral service aspects, such as the service interface and service interactions, whereas the concrete part binds the service to a specific protocol and endpoint, which is only used "pro forma", as JBI relies on its own messaging model that is independent of low-level communication details such as the protocol (e.g., SOAP) or the concrete endpoint address - services may specify only an endpoint name as target, and JBI's message router (see below) determines the concrete endpoint. This decouples the calling service (the Service *Consumer*) from the target service (the Service *Provider*); also more than one service can implement a particular WSDL interface, allowing for dynamic service invocation based on specifc endpoint properties (c.f. [JBI:26-27]). As a result, the caller need not know the exact target and whether it is available in the local environment or as a remote resource, which enables full location transparency.

Figure 4.4 shows a basic example of how an external event is routed through a JBI container: first it is received by a SOAP `BindingComponent` (maybe supporting *WS-Eventing*), which translates the incoming event into a normalized message and creates a new inbound JBI `MessageExchange` to let the normalized message router (NMR) route the message through the JBI environment. Because the BPEL `ServiceEngine` is configured to be part of the composite application, it receives the message and invokes a JDBC `BindingComponent` as part of a business process. The BindingComponent then transforms the normalized message to a JDBC Query, which is sent to a *CRM* database.

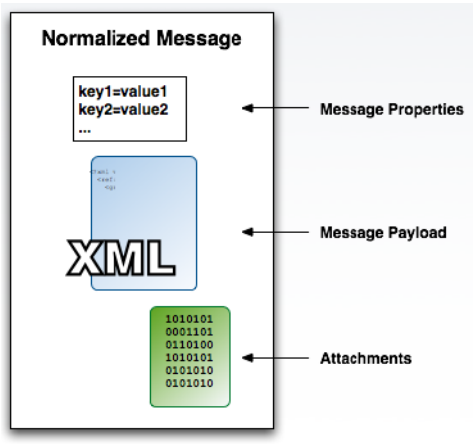


(from the presentation Practical SOA with Open ESB[4], slide 18)
Figure 4.4: Basic example of a JBI composite application processing an event

[4] http://www.objectware.no/OWFilesystem/filer/Practical%20SOA%20with%20Open%20ESB-WEB.pdf

Although JBI uses the same model for service description as web services, the integration model is very different from web-service integration (as described in Section 3.3.3.1): “WS standards only describe how a request is represented over the wire. They don't provide for any mechanism to ‘host’ services. JBI tries to fill this gap.” (from [Juric2007:304]). Also the messaging between components is firmly based on WSDL, using similar message exchange patterns (MEPs), which provide a way “to abstract communication away from implementation notions such as synchronous and asynchronous calls” [TrowBridge2004:150], standardizing a set of communication patterns that all services agree upon. This ensures flexibility in integration, abstracting from temporal coupling (which would impede location transparency), and provides a common understanding of *conversations*, a high-level service-oriented communication concept introduced in Section 3.3.3.

All messaging in a JBI environment is handled by a common messaging backbone, the NormalizedMessageRouter (NMR): “The NMR can be thought of as an abstract WSDL-defined messaging system infrastructure, where bindings [BCs] and engines [SEs] serve to provide and consume WSDL-defined services.” [JBI:21]. In the same source, the NMR is motivated as thus: “This mediated message-exchange processing model decouples service consumers from providers, and allows the NMR to perform additional processing during the lifetime of the message exchange.”. At the low level, the NMR implements message-based integration, while at a more abstract level, it provides a service-oriented communication layer. As a result, the NMR in particular and JBI in general are often seen as a specification for an ESB, but JBI defines a more general architecture, with an ESB being only one possible implementation of this architecture (see also Figure 3.10 in Section 3.3.3.2), as noted by Ron Ten-Hove: “JBI was deliberately crafted to support multiple approaches to building an ESB. This has resulted in some quite different approaches.”[5].



from the article Service Oriented Integration with ServiceMix[6]
Figure 4.5: JBI Normalized Message structure

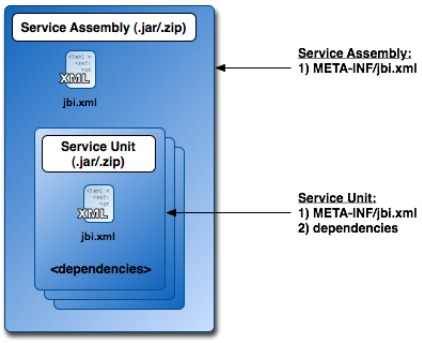
Messages are stored in normalized form (see Figure 4.5) and accessible through an associated API, which defines a `NormalizedMessage` in XML format, including several message properties (also called the *message context*), the message payload, and optional attachments, which is specified in [JBI:13]. It is important to note that the message format is *not canonical*, where all services have to agree on a common, concrete message definition, but only *normalized*, using XML and a common container format that specifies the aforementioned message parts. The message content is *service-specific* and openly specified by the service's WSDL. In this way, JBI's messaging model can be compared to e-mail: message properties (or metadata) serve a similar purpose as e-mail headers (“From:”, “To:”), which are used by mail servers (or the NMR) for routing and may also contain custom, service-specific information (X-Headers). The raw message payload is service-specific (the text only has to be

[5] from Ron Ten-Hove's blog entry Is JBI an ESB? [http://blogs.sun.com/rtenhove/entry/is_jbi_an_esb]
[6] http://servicemix.apache.org/articles.data/SOIWithSMX.pdf

understood by the recipient) and defined by the service's WSDL, in the same way as an e-mail's content is not parsed by transmitting systems and only subject to a common definition of semantics (i.e. a common language). Optionally, e-mails can also contain attachments which serve the same purpose as in JBI.

Finally, the JBI specification [JBI:c6,c10], declares a set of core infrastructure services for installation, deployment and management of components, artifacts and shared libraries, and for querying components and services, which resembles a service registry comparable to UDDI in a typical SOA environment. This management layer is specified as a set of JMX MBeans (as defined by the *Java Management Extensions* (*JMX*) specification [JMX]) that compliant implementations must provide, thereby enforcing a standardized management access across JBI implementations. This ensures that services can be set up and configured as needed for a particular task or use case, from any place, and that they can be reused in any JBI-compliant runtime. The absence of a standard management access was identified by Sun and others as a major drawback of the JEE 1.4/5 specification, which did not define any management layer. The result was that every JEE implementation realized management access differently, reducing interoperability between JEE runtime environments and complicating deployment and administration.

In JBI, components (ServiceEngines and BindingComponents) are installed using standard, JMX-based installation mechanisms as specified in [JBI:59]. This ensures that JBI components can be deployed into any JBI-compliant runtime in the same way (see Section 4.4.2). For configuring these components at runtime, the specification defines a standard packaging format, the `ServiceUnit`, which includes needed configuration files and optionally additional artifacts, and specifies the target component where these files should be deployed to. Several ServiceUnits are packaged inside a ServiceAssembly, which describes an entire integrated (“composite”) application, as illustrated below:



(from the article JBI Packaging in ServiceMix[7])
Figure 4.6: JBI packaging model

The JMX interface is covered in more detail in Section 6.4.2.4, where additional, custom `MBeans` are used to manage the JBI components developed for the prototype. The core management services are represented through yellow boxes in Figure 4.1 above.

To summarize, JBI can be defined as “a loosely coupled integration model for distributed services within a Service-Oriented Architecture (SOA)”[8] that provides an environment in which plug-in components reside, with interoperation between plug-in components using message-based service invocation described through WSDL, and a set of services to facilitate management of the JBI environment, by defining mandatory MBeans as part of a JMX-based management infrastructure. Although JBI runtimes may be implemented as an *Enterprise Service Bus*, this is not mandated by the specification, and implementors are free to chose alternative architectures that may be purely event-driven or resource-oriented (see also Section 8.5).

[7] http://servicemix.apache.org/5-jbi.html#5.JBI-JBIpackaging
[8] from The OpenESB Wiki [http://wiki.open-esb.java.net/Wiki.jsp?page=AOSD]

4.2.2. A Comparative Analysis of JBI

The following sections relate JBI to existing and complementing standards, showing differences to the new approach and the advantages provided for realizing the solution outlined in Section 4.3 below.

4.2.2.1. Relation to Event-Driven Integration

The concept of an event-driven architecture has been introduced in Section 3.3.5, where several relations between SOA and EDA are identified. As a result, also JBI allows event-driven integration by using corresponding ServiceEngines, such as the IEP SE, an open source complex event-processing engine used in OpenESB (see Section 4.4.2)[9].

Also, the JBI specification does not mandate a concrete implementation of the `NormalizedMessageRouter`. As a result, different message-based communication patterns may be realized. This can be either modeled as request/reply, as common in an SOA, but also as publish/subscribe, as used in message- or event-driven systems. Currently available JBI implementations mostly support more than one communication style in order to support a variety of integration solutions. For example, ServiceMix, the JBI implementation chosen for realizing the prototype, provides an advanced message queue (ActiveMQ) for reliable and fault-tolerant messaging, but also supports a staged event-driven architecture (SEDA, see Section 3.3.5 for a short introduction). Most JBI implementations also provide a JMS BindingComponent, which allows integration of existing messaging systems.

As a result, integration architects do not have to chose between SOA and EDA, but can freely apply both architectural styles as needed, thereby combining advantages of both approaches. [Balasbanmugam2008] provides an example that applies the concepts of an event-driven architecture to realize a business intelligence application for fraud detection, using the IEP event-processor engine provided by OpenESB (see below) running inside the open source JEE GlassFish application server.

4.2.2.2. JBI Compared to JEE and JCA

JCA (introduced in Section 3.3.7.1) has been the standard way to integrate external applications in the JEE world. Compared to the approach used in JBI, it has several deficiencies (c.f. [Christudas2008:56]), which result from JCA targeting deployment, not runtime management – for this, it relies on JEE application servers, which do not provide advanced integration facilities covered in JBI.

JCA realizes a low-level protocol-integration between the Adapter and an external system, but offers no means for integrated components to communicate with each other in a standard way, as there is no common messaging infrastructure. Although version 1.5 of the JCA specification added a “Message Inflow Contract” to enable `ResourceAdapters` sending messages to message-driven beans, possibilities for communication are limited and the resulting solution is more tightly coupled. A fully service-oriented approach as with JBI enables highly dynamic and peer-to-peer collaboration while at the same time ensuring transparency and loose coupling between components by building on a service oriented architecture and using related standards.

JBI relies on the WSDL standard and offers advanced message routing and translation capabilities, using a common XML-based message-format (the NormalizedMessage) and a common message routing backbone (the NormalizedMessageRouter). Capabilities can be added by installing additional ServiceEngines for advanced message transformation, enrichment, or encryption, and additional external resources can be integrated by adding BindingComponents for, e.g., SNMP, FTP, CICS, or packaged applications like SAP, databases like Oracle etc.

[9] the Wiki page Event Driven Architecture in Open ESB [http://swik.net/GlassFish/The+Aquarium/Event+Driven+Architecture+in+Open+ESB+-+ESBs+ain't+just+for+SOA+anymore/cc0x2] gives a good overview and provides further reading on the subject

Also, maintenance of JCA Adapters is more complicated than with JBI: JCA does not fully specify a common packaging format that allows auto-deployment[10]. While the JCA specification defines ResourceArchives (RAR), it only demands an informal (though XML-based) description and leaves much room for implementors, both application server vendors and application developers. The option to deploy ResourceAdapters in an application server or standalone adds even more room for proprietary extensions and results in more work on installation and updates. JBI specifies a JMX-based management layer and offers MBeans for installation, maintenance and runtime-configuration of components. Individual components may also offer extended configuration by using a standard ConfigurationMBean, allowing developers to implement advanced integration solutions without limiting ease of administration: components can be deployed automatically by using hot folders or any JMX-compliant management console.

To summarize, Table 4.1 shows the relations between JBI and existing Java EE standards when viewed from a conceptual perspective.

JEE concept	**JEE**	**JBI Equivalent**
Web tier	Web Archive (WAR)	– (JBI does not include any web presentation besides indirect web access through MBeans)
Service tier	Enterprise Java Bean (EJB) (statically configured with packaged descriptors)	ServiceEngine (dynamically configured through descriptors and artifacts in a matching ServiceUnit)
Composite Application	Enterprise Archive (EAR)	ServiceAssembly
Integration of External Components	Resource Adapter (RAR) (through JCA)	BindingComponent
External Communication	based on contracts (as defined in the JCA specification)	message based mediation (internally using a canonical message format, external format is proprietary and system-dependent)
Management	- (JBoss offers a proprietary concept of ServiceArchives (SARs) based on JMX MBeans)	ComponentMBeans (JBI specification mandates a set of standard JMX MBeans)

[a]see also [JEE5Tut:53], *Packaging Applications*

Table 4.1: Conceptual relations and overlap between JBI and JEE

The relation of JBI and existing standards in the Java EE world from an architectural perspective is summarized by the JBI specification lead Ron Ten-Hove as follows:

> JBI works at a different level than Java connectors. JBI allows services and protocols to interoperate using a WSDL-described model, whereas Java connectors provide a Java-centric way of interacting with EISs. Java connector architecture fits into JBI as a particular type of binding component. JMS also fits into JBI this way.
>
> —Ron Ten-Hove (answering a question in Sun's JBI forum)

4.2.2.3. Relation to SCA

The *Service Component Architecture* (SCA) is a relatively young standards effort originating from IBM, but contributed to *OASIS* as an open standard under governance of the OpenCSA board (Open Composite Services

[10]e.g. JEE application servers use different deployment schemes for shared libraries, resulting in ClassLoader-issues during deployment

Architecture), which defines SCA as “a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture (SOA). […] SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing application function within SCA compositions.” [OpenCSA2008].

A good introduction to the standard is given in [Chappell2007], who defines the motivation behind SCA as “a way to create components and a mechansim for describing how those components work together.”. An important aspect of SCA is language and platform independence, as SCA only defines a common assembly mechanism. Language bindings provide the mapping from specification to a concrete implementation. Currently there are mappings for C++, Java (including specifications integrating EJB, JMS and JEE), PHP, BPEL and Spring, web services and others. The specification is currently at version 1.0[11].

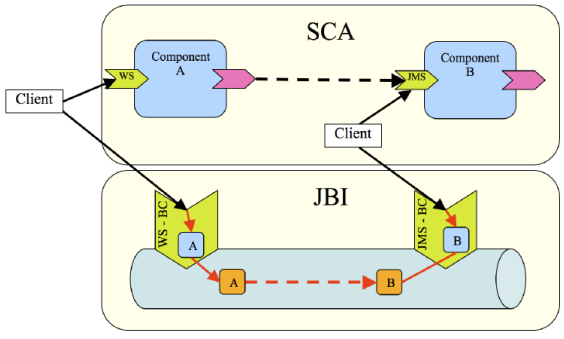
While SCA is often seen as an alternative or competitor to the JBI standard[12], both approaches are actually complimentary, as they take a different perspective on integration, but both build on service-oriented integration and provide a loosely-coupled component model (c.f. [OSOA2007]). SCA concentrates on the architectural aspects of designing composite applications out of service components (hence the name), whereas JBI standardizes an integration infrastructure and defines a runtime environment where services are integrated and communicate over a message bus. Thus, a composite application could be designed as an SCA model and deployed into a JBI runtime. This is the approach currently taken in the Eclipse Service Tools Platform, which implements a mapping from the SCA design model to JBI's runtime model. The convergence of JBI and SCA is demonstrated with an example in [Mos2008], who suggests the following mapping of SCA artifacts to JBI counterparts:

SCA artifact	**JBI counterpart**
Component	ServiceEngine
Composite	ServiceAssembly
Binding Type	BindingComponent
Wire	Configuration in ServiceUnits, or MessageExchange

[a](according to [Mos2008])

Table 4.2: Relation of JBI and SCA

This relation is also illustrated in Figure 4.7 below:



(source: [Mos2008], originally taken from the Eclipse STP Wiki [EclipseSCA])

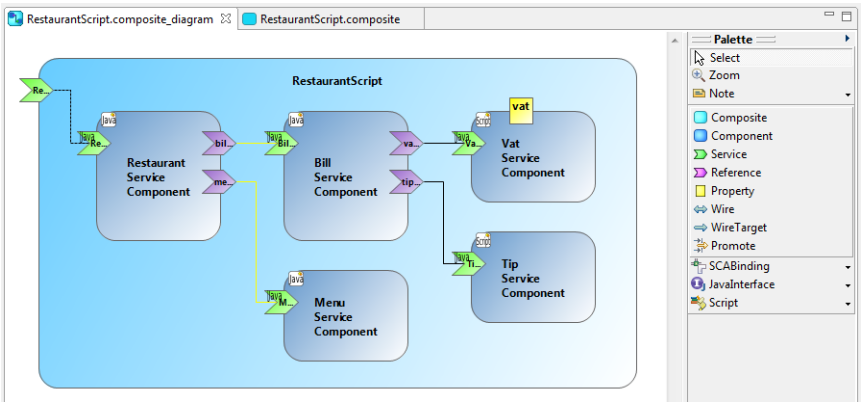
Figure 4.7: Mapping from SCA to JBI

[11]Unlike other specifications, the SCA specifications are quite compact and written in a straight forward manner, making them approachable with reasonable effort…

[12]This perception is led by the standards coming from two competitors in the industry, IBM and Sun, and from the fact that IBM (and BEA) sustained from its vote on JBI and left the corresponding JCR expert group.

Current implementations include Apache Tuscany, which is introduced together with SCA in [Christudas2008], [Fabric3] and [SCOrWare].

Tool support for designing SCA composite applications is currently provided mainly by the Eclipse Service Tools Platform, with the SCA subproject [EclipseSCA], which is part of the official Ganymede release. The editor is shown in Figure 4.8 below:



(source: [EclipseSCA])
Figure 4.8: Eclipse STP SCA Editor

An important limitation that makes SCA unsuitable for tool integration is that SCA is essentially limited to a *single domain* or runtime implementation of a single vendor. There is no support for wiring together SCA-composites across domains; they can only interact using common interoperable protocols such as web service communication. This is explicitly noted in [Chappell2007], who clarifies that “an SCA application communicating with another SCA application in a different domain sees that application just like a non-SCA application; its use of SCA isn't visible outside its domain.”. As the communication protocol among SCA composites is implementation-dependent and not standardized, components *cannot* be interchanged between different runtime implementations (e.g., Fabric3 and Tuscany), which inhibits component reuse across SCA implementations. This is a serious and unacceptable limitation for the proposed tool integration framework, besides the complete absence of needed ESB-like infrastructure services in the SCA specification, which results from the standard's focus on a design-time architecture.

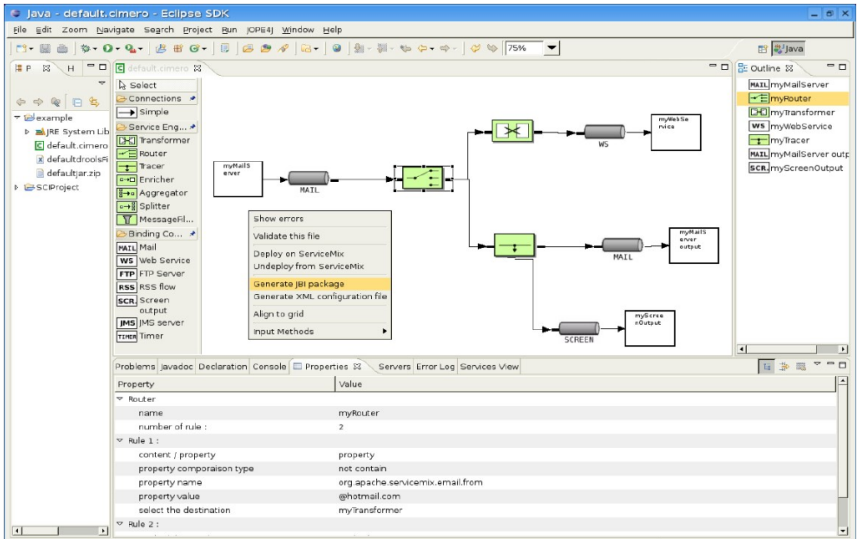
To summarize, there is much potential in combining the two standards, taking the advantages of language independence and a very straightforward, easy to use API from SCA, and utilizing JBI's runtime interoperability and rich integration infrastructure, which is already implemented in several ESB-like solutions. Where the two standards overlap, the *JBI 2.0* specification [JBI2], JSR-312, aims to differentiate the two and clarify any ambiguity. Also, tool support for both solutions is quickly improving, with Eclipse providing support for designing SCA solutions (which indicates the roots of both projects), and NetBeans providing rich JBI support. In Chapter 8, SCA will be shortly revisited together with the related Service Data Objects (SDO)-standard for high-level data integration.

4.2.3. Development and Tooling Support

While it is possible to develop JBI components and ServiceAssemblies without IDE support by using XML editors and writing deployment descriptors and configuration files by hand, additional tool support greatly improves development productivity and reduces the likelihood of errors that may be hard to find later, given the rather complex configuration of JBI. An overview of the steps involved in developing custom JBI components is given in [Kieviet2007], who also examines current tool support.

The first level of tool support is provided by using Maven2 archetypes provided by Apache ServiceMix[13], which are essentially project templates for the popular Java build tool. These can be used, e.g., in Eclipse using the Maven integration plugin. Sadly, no visual editors are provided and configuration has to be done by hand. In ServiceMix, this is eased by providing a lightweight XML-based format for configuring JBI components and assemblies (using XBeans with `xbean.xml` configuration files), and by supporting the deployment of POJOs (Plain Old Java Objects) as JBI components through ServiceMix XBeans.

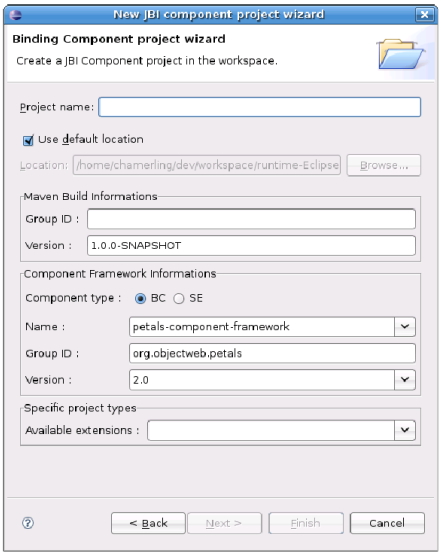
Advanced and visual configuration of JBI assemblies is now supported as part of the Eclipse ServiceToolsPlatform (introduced in Section 3.2.4.2.1), developed as a separate project originally called CIMERO. The editor (shown in Figure 4.9) allows rapid development of integration solutions using existing integration components, following enterprise integration patterns (using the patterns and symbols introduced in [EIP]). The resulting configuration can then be deployed to Apache ServiceMix or PEtALS ESB, two JBI implementations covered in Section 4.4.2.



(taken from [EclipseSTP], EID subproject)
Figure 4.9: Enterprise Integration Patterns in Action using Eclipse STP's EID editor

Unfortunately, neither CIMERO nor the ServiceToolsPlatform support the development or use of custom JBI components, but only provide a set of preinstalled messaging and routing components. Integrating custom components is supported by using extension points provided by the editor, but this requires manual coding and configuration and is more targeted at STP/CIMERO developers. Currently, full development support for visual creation of composite applications is only provided for the SCA standard, which is shortly covered in Section 4.2.2.3. Recently, support for adding JBI components to a SCA assembly has been added, and the resulting composite application can be deployed to a JBI runtime, such as ServiceMix or PEtALS, but no full design time support (e.g., for developing ServiceEngines) is yet provided for JBI.

[13]see the tutorial Using Maven to develop JBI applications [http://servicemix.apache.org/2-beginner-using-maven-to-develop-jbi-applications.html]



(taken from [EclipseSTP], EID subproject)
Figure 4.10: Eclipse STP editor with JBI support

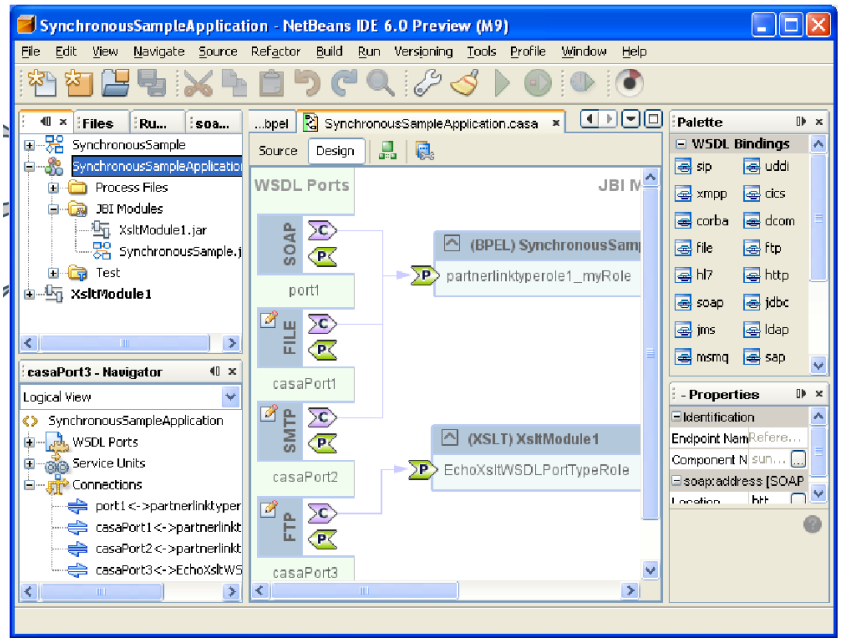
[Swordfish] is an emerging open source project donated by SOPERA (introduced in Section 3.2.4.2.1). The project aims at providing a runtime integration infrastructure based on OSGi, using JBI for protocol and service integration, but is not yet available in open form and does not yet provide any design time support; for this, it relies on the aforementioned ServiceToolsPlatform.

FUSE (see Section 3.3.3.3.2) is a commercial solution that builds on ServiceMix and provides additional features and tool support, e.g., the recently introduced visual integration designer[14]. Being commercial, it could not be used for the proposed solution, and support for developing custom components is still limited.

The NetBeans Java IDE provides design and runtime support for creating and deploying composite JBI applications from within the IDE, through the CASA (Composite Application Service Assembly) visual editor, associated project types and related wizards[15], illustrated in Figure 4.11 below.

There is also a commercial variant called JavaCAPS[16] that provides a full integration stack for enterprise customers, including the Glassfish application server and management tooling. While the CASA editor provides rich support for configuring composite applications using existing components, support for developing custom plugins is limited to a wizard that creates the necessary skeleton and configuration, and simple deployment testing. For integrating custom components into the CASA editor, it is necessary to develop a NetBeans plugin, which is a documented but tedious procedure[17]. Also, JBI development with NetBeans is tied to the Glassfish ESB (see Section 4.4.2.1 below), although components developed with NetBeans can be deployed into any JBI-compliant solution.

[14]for an analysis on the mixture of open source and commercial ESBs, see the article Open source/commercial ESB hybrid reflects SOA reality [http://searchsoa.techtarget.com/originalContent/0,289142,sid26_gci1285526,00.html?track=NL-110&ad=615708&asrc=EM_NLN_2731874&uid=774931]
[15]see the OpenESB Wiki page on JBI component development support [http://wiki.open-esb.java.net/Wiki.jsp?page=JbiComponentDevTools] and the official NetBeans SOA home [http://www.netbeans.org/features/soa/index.html]
[16]see Sun's web page on JavaCAPS [http://developers.sun.com/javacaps/]
[17]e.g., see Chad Gallemore's blog entry Creating a Binding Component Deployment Plug-In for Netbeans [http://gallemore.blogspot.com/2007/05/creating-binding-component-deployment.html]; according to [Kieviet2007], there is now a wizard to achieve the same



(originally taken from [Kieviet2007])
Figure 4.11: Developing composite applications with the NetBeans CASA editor

ChainBuilder provides a visual JBI component development platform based on Eclipse. Like the aforementioned solutions, it comes with a designer for constructing ServiceAssemblies and for configuring individual components. Unlike those solutions however, ChainBuilder also provides an API and necessary tooling for developing and deploying custom JBI components, which can be easily integrated into the visual editor, thus enabling a coherent visual design and development workflow for efficiently realizing custom integration solutions. For these reasons, ChainBuilder was used for realizing the prototype, as shown in Section 6.4.4.

The current tooling landscape reflects the evolutionary state and enterprise-centric focus of the JBI 1.0 standard, providing integration architects with a way to compose new, integrated applications out of existing services and Adapters available from third parties. It does not yet provide a comfortable environment for individual developers that need to create new Adapters for custom solutions.

4.3. Using JBI for Tool Integration

"The application, in effect, becomes a microcosm of the web, in that it's using many of the same composition techniques internally that the network is using externally."
--Mike Hapner in an Interview about SOA best practices[20]

As mentioned in Section 4.2.1, JBI provides standards-based mediation and composition of existing components by providing a high level abstraction mechanisms and by integrating various container and protocol standards, as often encountered in typical enterprise integration scenarios. This makes it possible to design an integration architecture which combines previously incompatible protocol and data formats, as often the case in heterogeneous tool landscapes, in a loosely-coupled way. *JBI* solves existing issues in legacy integration, as outlined in Section 2.2.1, by providing a service-based façade (WSDL), enabling integration of existing applications into a service-oriented, standards-based integration middleware, avoiding the need for implementing yet another custom integration framework with proprietary container and communication formats. It is therefore possible and

[20] http://java.sun.com/developer/technicalArticles/Interviews/hapner_qa.html

feasible to use *Java Business Integration* concepts for legacy integration, e.g. BindingComponents for client-side tool integration, even if this is not the original target domain of the enterprise-oriented JBI standard. The wide scope of application is also noted by the "inventor" of JBI:

> The ability of JBI to integrate disparate applications as WSDL-described services is only limited by the expressive power of WSDL itself, and the ability of the component author to map the application to such a model.
>
> —Ron Ten-Hove, JBI specification lead, in an interview [Sommers2005]

Integration of existing applications in the enterprise domain faces similar challenges as integrating off-the-shelf tools on the client side. By using JBI's concept of BindingComponents, any tool that can send or receive messages or function calls over an external interface can be integrated. This enables previously isolated tools to consume or provide services to other framework components (and so other tools) in a transparent manner. For a tool to be integrated, only a BindingComponent has to be developed that translates between the proprietary tool interface and JBI's normalized message format. This enables transparent data integration without having to change existing tools or investing effort in developing models for mapping the tool's data model to a common representation, which can be problematic as shown in Section 3.3.6. Functional integration at the logical level is realized by implementing a corresponding ServiceEngine that exposes the tool's functionality as a set of services inside the JBI environment.

4.3.1. Tools as Composite Applications

In the context of JBI, the term *composite application* is often used to describe the desired outcome of an integrated application, which is composed of existing (where possible), independent service components that are combined in new ways to solve current (business) needs. More precisely, it can be defined as "a collection of existing and independently developed applications and new business logic orchestrated together into a brand new solution of a business problem that none alone can solve."[19]. A good overview is also given in [Altman2007].

Viewing a set of tools as a *composite application* leads to a holistic perspective on tool integration, where individual tools grow together, increasingly overlapping in functionality and data, resulting in a transparent but significant advance in user experience:

> A composite programming model has also interesting consequences in terms of "application boundaries": there are no visible technical or physical boundaries, only logical ones. A composite programming model typically exhibits a federated and collaborative point of usage where users can initiate, work on and complete any number of user activities irrespective of the information services or business processes they participate in. This point of usage can even be different for different users and support clients of any type (mobile, desktop…) more easily. In other words, different user activity containers may implement the same user tasks.
>
> —from the book *Composite Software Construction* [Dubray2007:22]

[Raj2007] shows how *Java Business Integration*-standard can be applied to process integration in an enterprise setting, using a *WS-BPEL* ServiceEngine: "Service-enabled applications create the opportunity to compose functions from disparate and cross-functional applications to model business processes that transcend application and enterprise boundaries."

Figure 4.12 provides an overview of the different dimensions in application composition, dividing the integration landscape into four quadrants, spawning a diagonal line of evolution from statically designed, monolithic enterprise backend systems to highly dynamic and user-centric mashups on the web. Enterprise 2.0 applications (e.g., portals) can be viewed as frontends to carefully integrated backend systems, whereas composite applications

[19] from the OpenESB Wiki page on Composite Applications [http://wiki.open-esb.java.net/Wiki.jsp?page=CompositeApplications]

are dynamically combined out of existing backend services and are mainly used to realize integrated business processes.

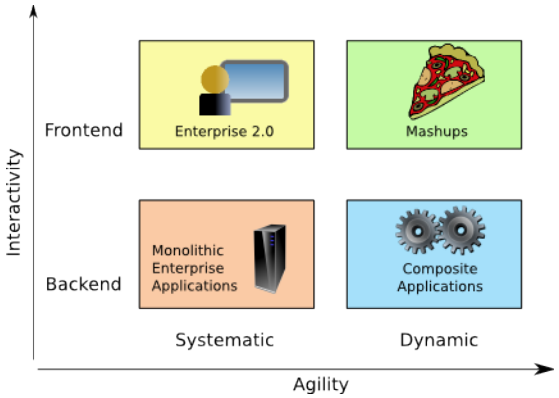


Figure 4.12: Dimensions and dynamics of application composition

Returning to the comparison of enterprise integration solutions to mashups from Section 2.1, this relation, which may seem far-fetched at first, is already implemented in the form of an Enterprise Data Mashup ServiceEngine[20] that facilitates development of mashups, e.g., for combining data from different databases or spreadsheet files. This shows the wide area of integration possibilities provided by JBI, which is of great advantage in tool integration.

While JBI is a Java-based integration infrastructure, it is not limited to integrating Java applications: being based on open industry standards such as WSDL, JBI allows integration of non-Java applications, web services and legacy systems through BindingComponents. Additionally, semantic integration is provided by using common integration services realized as ServiceEngines, as demonstrated by the prototype in Chapter 6.

4.3.2. Evaluation

A closer evaluation of JBI based on the requirements outlined in Section 4.1 shows that JBI fulfills the key requirements of *generality* and *flexibility*, as JBI is designed to accommodate heterogeneous enterprise environments with greatly varying needs and challenges, from proprietary backend systems to modern service-oriented applications. As a Java standard, JBI solutions are inherently *portable* to a wide array of platforms, which also facilitates the realization of uniform user interfaces, by using cross-platform UI frameworks like Swing, SWT or Java-based application platforms like Eclipse RCP (see Section 3.2.3), and thus fulfills the requirement of *homogeneity*.

BindingComponents support several requirements for tool integration: they realize *compatibility* and *COTS integration* as mentioned above, they provide *adaptability*, *extensibility* and *modularity* because they enable loose coupling of tool interfaces to the integration framework, and they translate from a tool's proprietary data model to the common, normalized message format used on the *JBI* message bus.

ServiceEngines, on the other hand, provide the necessary *functional integration*, which is abstracted from the original tool interface by a corresponding BindingComponent, and also support the realization of higher-level services for *rich data integration*, such as managing Relations or other general framework services like session or user management.

The NormalizedMessageRouter supports instantaneous (corresponding to *data timeliness*), *asynchronous* and *reliable* communication, whereas several implementations exist that allow fine grained weighting between various communication requirements.

[20] see the Glassfish Wiki page Enterprise Data Mashup [http://wiki.open-esb.java.net/Wiki.jsp?page=EnterpriseDataMashup]

Regarding *transparency*, JBI provides the necessary abstraction from communication details, protocol formats and framework internals, and is designed according to the needs of integration developers by providing a rich API that covers several important aspects of application integration. The only "overhead" introduced is the mandatory use of WSDL and XML messaging, but with the advantages this brings, it is a minor cost to pay. Once suitable BindingComponents and ServiceEngines are provided (for many use cases these do already exist), integration architects can now rely on tooling support in several forms (see Section 4.2.3), which provides the needed *flexibility* in building or adjusting integration solutions.

For end users, the solution is fully transparent, but *presentation integration* still has to be implemented by hand. Existing tools have to be extended with suitable interface code, so that users gain access to additional integration functionality provided through BindingComponents and ServiceEngines. This "last mile of integration" cannot be bridged by JBI alone, but the necessary foundation is already provided. Custom interface code in the integrated tool can now communicate with an accompanying BindingComponent in a straightforward and lightweight manner, without the need for custom integration code bound to the framework. It is sufficient to implement a channel for sending and receiving commands and associated data, e.g., using `TCP/IP`, as is done in the prototype. Communication with the integration backbone and API integration is fully handled by the BindingComponent, which provides all necessary mediation and further translation functionality (see also Section 6.4.1.3).

Because JBI also specifies a management infrastructure based on the JMX standard, a JBI based integration solution is inherently exposed for local and remote management using any management interface supported by the JMX implementation (e.g. SNMP for legacy management applications, HTTP for web based, distributed management, etc.).

Cross-cutting functionality like security can be realized through suitable technology selection, e.g., by selecting a message bus implementation that supports secure transmission and encryption of messages, or by implementing a ServiceEngine that validates user roles and privileges[21]. Scalability is supported by using a JBI implementation that supports distribution, e.g. PEtALS (see Section 4.4.2 below).

Concluding, JBI provides a modern and standards-based solution that provides service-oriented integration of previously isolated applications. Although being targeted at enterprise integration, there is a considerable overlap in challenges and a striking similarity in the problem domain with typically isolated and heterogeneous application landscapes in tool integration, which makes JBI a perfect match for tool integration on the desktop. This rationale is also supported by [Touzi2007]: "Service Oriented Architecture (SOA) seems to be the perfect support for applications interoperability […]. The chosen technical framework is based on a SOA implemented by a JBI (Java Business Integration) compliant ESB (Enterprise Service Bus)."

4.4. Realization

The proposed tool integration solution is entirely built on the foundation of the JBI standard, using custom JBI components to integrate COTS tools: BindingComponents provide the needed low-level integration and realize bridging from proprietary tool interfaces and protocols to normalized messages used in the JBI environment. Where existing web service interfaces or standards-based Adapters (e.g. JCA Adapters) are available, they can be reused as there are already corresponding BindingComponents available for various JBI implementations[22].

Semantic integration (e.g., data transformation, finding related tools, routing information to interested tools) and common tool services (e.g., common services for object and data manipulation supported by several Adapters) are implemented as ServiceEngines, which rely on BindingComponents to deliver and send data and to perform

[21]The selected JBI implementation, Apache ServiceMix, supports authentication, authorization and transport security using available standards such as JAAS or WS-Security, see the ServiceMix Security page [http://servicemix.apache.org/security.html]

[22]Although standards-conformant JBI components should work on every JBI implementation, there is a certain overlap in the market, and often more than one implementation is available for a particular Binding, e.g. HTTP, FTP or File.

corresponding actions in the target tools. The separation of low-level protocol integration from higher-level semantic integration results in a highly adaptable tool integration solution that can be easily adapted to different usage scenarios as needed, enabling dynamic workflows across heterogeneous and isolated tools.

The general design of the proposed solution is illustrated in Figure 4.13 below, using familiar icons from [EIP] to demonstrate the consequent use of enterprise integration standards and patterns. The general design using Adapters and Translators may seem reminiscent of Figure 3.15 in *Tool Integration Patterns* [Karsai2003] or even the existing ToolNet architecture, but it differs in several key aspects: it does not rely on model-integration and hence is not limited to data integration, it does not use low-level or proprietary communication in the messaging backbone (JBI's NormalizedMessageRouter instead of CORBA[23] or a custom messaging backbone), and there is a common, standardized API for the components. *ToolAdapters* are realized as JBI *BindingComponents* (in this figure, the DOORS BC communicates with a Doors Adapter – a set of tool-specific scripts – inside the DOORS application). Semantic translation is realized through *ServiceEngines*, which convert between tool specific commands and standardized, common service requests (similar to ToolNet Services). The manager or desktop component is represented by a JMX-based management console which holds tool-specific MBeans exposed by the ServiceEngine (e.g., data objects or tool functionality). Metadata is handled in a standards-based way using existing metadata fields in NormalizedMessages. Finally, workflows can be implemented by using an existing BPEL ServiceEngine, but this is not shown here for the sake of clarity.

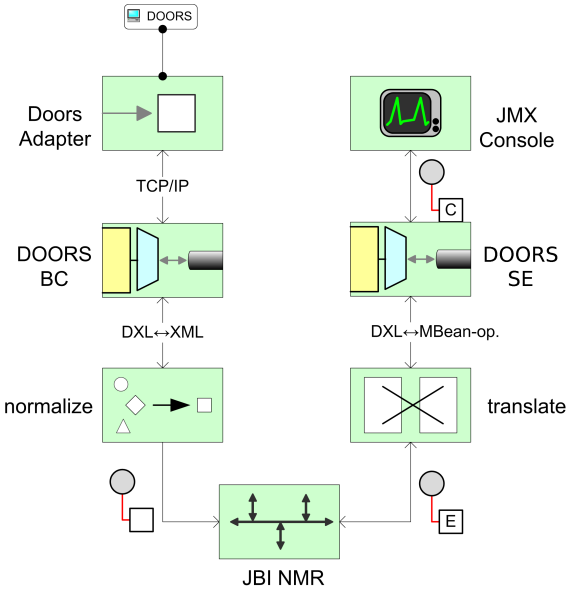


Figure 4.13: Design of the proposed solution

Using the JBI standard as a foundation for tool integration not only provides a common API and reusable infrastructure, but also enables to build on existing standards in enterprise integration, from backend integration standards like JCA to service-oriented integration using web services or REST, up to process-integration standards like BPEL.

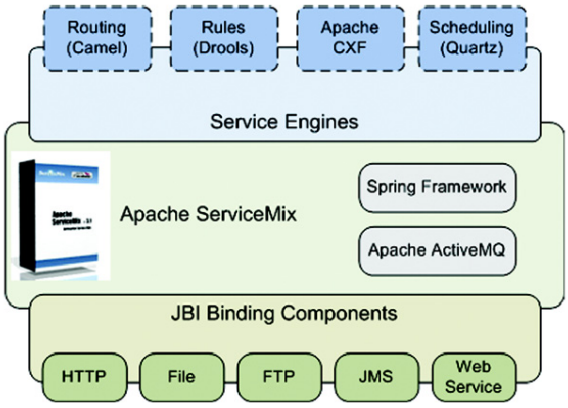
The following sections provide a short evaluation of available JBI implementations, beginning with the one that was chosen for the prototype implementation, Apache ServiceMix.[24]A good starting point for evaluating ESBs is also provided in [Rademakers2008:23], which focuses on Mule and ServiceMix and shows how different integration scenarios can be implemented using these ESBs, and [Christudas2008:63], who provides some background on JBI and service-orientation in general, using Apache ServiceMix for practical examples.

[23]The solution in [Karsai2003] is not bound to a specific middleware, but CORBA was the predominant middleware standard in 2003 and JBI not available then.
[24]see also the ServiceMix page How to Evaluate an ESB [http://servicemix.apache.org/how-to-evaluate-an-esb.html]

4.4.1. Apache ServiceMix

"Apache ServiceMix is an open source ESB (Enterprise Service Bus) that combines the functionality of a Service Oriented Architecture (SOA) and an Event Driven Architecture (EDA) to create an agile, enterprise ESB."
--from The Apache ServiceMix homepage[27]

Apache ServiceMix [Christudas2008:57] realizes a modular integration infrastructure completely based on the JBI specification. All components are based on the JBI standard and can be exchanged or adapted as needed, even the message router itself can be configured to custom needs: depending on requirements and constraints – such as latency, scalability or reliability – different message flows can be configured. Most message flows are implemented using Apache ActiveMQ, an open source JMS implementation. ServiceMix comes with a variety of ServiceEngines for message transformation, routing or security, and dedicated BindingComponents for supporting protocols like `TCP/IP` or JMS. Where possible, existing open source implementations (mostly from other Apache projects) are reused, e.g. Apache CXF for web service integration, Jencks for integrating JCA Adaptors, or Apache Camel for dynamic routing based on a simple DSL (see also Section 8.3). Figure 4.14 shows the ServiceMix architecture and some components available:



(from [Rademakers2008:25])
Figure 4.14: Apache ServiceMix architecture overview

Development tooling is provided through Ant tasks and Eclipse project templates, using Maven archetypes for building BindingComponents and ServiceEngines. A visual designer is available (CIMERO, now part of the Eclipse STP project), but as mentioned in Section 4.2.3, ChainBuilder provides a more complete solution that uses ServiceMix as the underlying JBI implementation.

There is a rich community and developer support around ServiceMix, and the development community quickly adopts and incorporates new standards and solutions in service oriented and message based integration, such as JBI 2.0 [JBI2] or OSGi with the upcoming version 4 (see Section 8.1 for more on the future of ServiceMix and JBI). An excellent introduction to JBI, ServiceMix and their combination is given in [Snyder2007], who also provides some insight into the next generation of JBI and ServiceMix.

Several other solutions use ServiceMix as a foundation, including a commercial variant, the FUSE integration suite, which is a pre-packaged and advanced ("enterprise-hardened") version of ServiceMix with additional tooling and enterprise-level support. [GASwerk] is an open source solution that provides an integrated SOA stack based on Apache ServiceMix, using ApacheCamel for implementing dynamic rules-based routing, Geronimo as the underlying application server, JMS for distributed messaging and Spring for lightweight configuration.

[27] http://servicemix.apache.org/home.html

ServiceMix was selected for the prototype implementation in Section 6.4.1.2 because of the open source nature combined with good developer support (Wikis, newsgroups, articles) and fidelity to the JBI specification. The project, being an Apache effort, enjoys high visibility and support in the open source Java community and is progressing steadily towards stability and standards support. While developing the prototype, it was accepted as a top level Apache project, which is a strong sign of commitment, stability and viability, making a strong foundation for building an integration solution on. Missing high-level tool support, esp. visual design of ServiceAssemblies and support for implementing custom components, is compensated by independent projects that use ServiceMix as a runtime, and add the necessary tooling and additional API support. Because FUSE is a commercial project, and Eclipse STP does not yet support the full JBI specification, ChainBuilder was used for designing and implementing the proposed solution.

4.4.2. Alternative Implementations Considered

Several implementations were evaluated for applicability to the requirements outlined in Section 4.1, esp. for tool integration using custom components, as necessary for realizing the prototype. All solutions presented here are open source and based on the JBI standard, with the notable exception of Mule, which only provides an external JBI binding and does not support JBI ServiceEngines. Mule is nevertheless included as a reference to widen the scope of this short open source ESB comparison, and because of its popularity[26]. In the following sections, the solutions considered will be introduced, and at the end, a feature matrix is provided for easier comparison of the individual ESB's support for the key criteria evaluated.

4.4.2.1. Glassfish and OpenESB

Project [OpenESB] is Sun's reference implementation of JBI and is also used in the open source GlassFish JEE application server (which serves as a reference implementation for the JEE specification). There is a central community hub (the OpenESB wiki) where already many JBI components are available, also for integrating legacy applications like SAP or protocols like CICS. Rich visual tooling is provided as part of the NetBeans “SOA pack”, which integrates Glassfish as a JBI runtime environment and supports design and deployment of ServiceAssemblies from within the IDE, as illustrated in Figure 4.11 above.

There is growing support for developing custom components, but their integration into the visual editor is not as simple as with ServiceMix (using the ChainBuilderIDE), which also has broader community and developer support, both in terms of documentation and help, but also in terms of practical experience and installed user base. Also, ServiceMix is more an independent project, whereas OpenESB is still closely tied to Sun.

4.4.2.2. PEtALS

A lesser known but advanced JBI implementation is provided by the OW2 consortium: [PEtALS][27](“the European open source ESB”) has been designed for enterprise-grade deployments with support for distribution, security and modularity at its core, and a rich web-based management and instrumentation interface[28]. The container is based on a custom, language-neutral component model (“Fractal”[29]), which allows porting to other platforms (e.g., embedded systems using the C implementation). PEtALS is used in several large-scale deployments (e.g., for the French social security system bank ACOSS) and in the research project SOA4All[32], which aims at pro-

[26]JBossESB [https://www.jboss.org/jbossesb/] is the latest addition to the stack of open source ESBs. It was not available when the prototype was realized, and JBI support is equally limited as in Mule, which is much more wide spread and mature. For these reasons, it is not included in this survey.

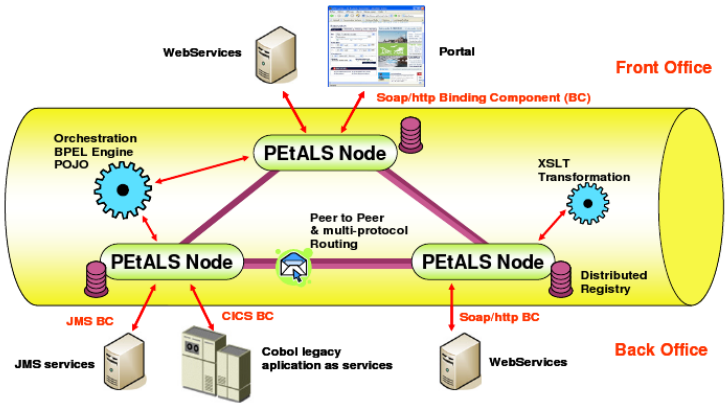
[27]documentation is also available at ebmwebsourcing [http://www.ebmwebsourcing.com/produits/documents.html], the company behind PEtALS

[28]recently, support for multi-node configuration using JASMINe [http://wiki.jasmine.objectweb.org/xwiki/bin/view/Main/] was added

[29]see the Fractal homepage [http://fractal.objectweb.org/]; the name suggests a recursive component model, where components are composed out of components that expose several services, a model that fits well with JBI's composite application concept

[32] http://www.soa4all.org

viding a semantically integrated “web of services”, merging SOA with the WWW. The basic architecture is shown in Figure 4.15 below:



(from Case study : deploying PEtALS on a national scale (ACOSS)[33])
Figure 4.15: PEtALS ESB Architecture

The *Component Development Kit* (CDK) provides a JBI-compatible extended API that eases development of components. Tooling is provided via an Eclipse plugin[32] that originates from the CIMERO editor (see Section 4.2.3) and supports configuration and deployment of ServiceAssemblies. Integrating custom components in the visual designer requires manual coding. Community and developer support is limited, and it is hard to find additional information for PEtALS outside the project's web site; also, downloading PEtALS requires registration, although the project itself is open source.

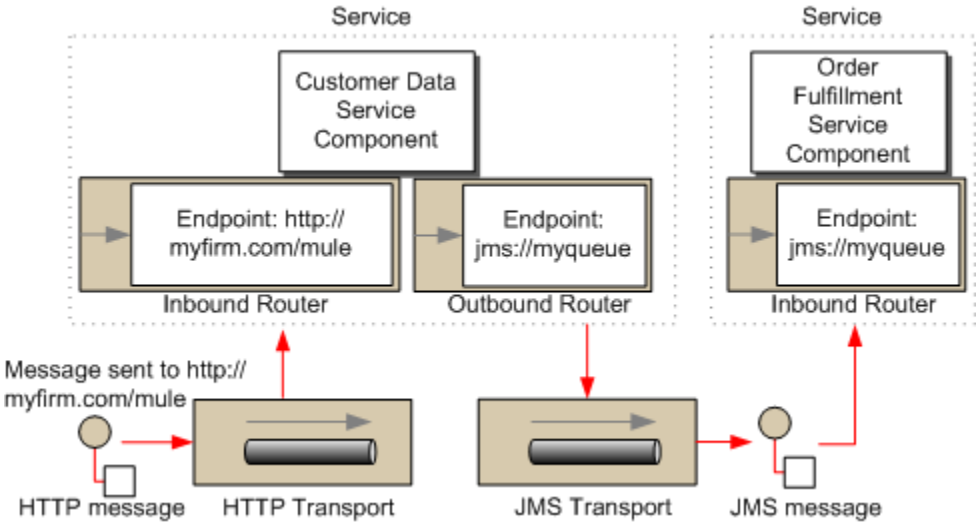
4.4.2.3. MuleSource Mule

As mentioned in the section's introduction, [Mule] plays a special role, as it is not a full JBI implementation, but only provides external JBI connectors, which can be used to integrate with other JBI environments like ServiceMix. It is also not a typical ESB, as is often stated, e.g., in [Menge2007], because it does not provide a common messaging backbone and associated backend services, but more a freely combinable mediation layer where components are not plugged *in* but sticked together as needed[33]. This is seen as an advantage over traditional ESBs, stating that “One difference between Mule and a traditional ESB is that Mule only converts data as needed. With a typical ESB, you have to create an adapter for every application you connect to the bus and convert the application's data into a single common messaging format. […] Mule increases performance and reduces development time over a traditional ESB.”

As a result, Mule uses a proprietary API and message format and does not embrace JBI or SCA as its underlying integration model. *Service components* (previously called *Universal Mule Objects* or UMOs) implement custom application logic and operate on the content of a message on a logical level, similar to JBI's *ServiceEngines*: they are decoupled from specific messaging or protocol formats required for communicating with other components or external applications. This is performed by *Transformers*, which convert between different data formats and wrap them in Mule messages, whereas *Transports* handle protocol conversion as needed, bridging between communication protocols such as `JMS`, `SOAP`, `HTTP`, File or proprietary protocols, similar to JBI's *BindingComponents*. Finally, *Routers* are used to deliver incoming messages to Service components and for sending outgoing

[33] http://www.ebmwebsourcing.com/images/stories/media/ow2-casestudy_petals_acoss.pdf
[32] see PEtALS-Eclipse [http://www.ebmwebsourcing.com/projects/petals-eclipse/]
[33] The *Mule Getting Started Guide* [Mule], chapter *Understanding the Messaging Framework*, itself states that “Mule is *based* on *ideas* from Enterprise Service Bus (ESB) architectures.” (emphasis added), making clear that not all ESB principles are implemented or supported. See also the interview with IONA [http://blogs.zdnet.com/open-source/?p=2286], where Mule is more related to ApacheCamel than to an ESB, being only a part of the integration puzzle.

messages to the target component according to the configuration. In JBI, there is a central, shared router that handles messaging and connects all endpoints, the *NormalizedMessageRouter*. Routers can perform additional logic such as filtering or message composition/decomposition, implementing the message routing-patterns from [EIP:225]. The basic architecture and message flow is illustrated in Figure 4.16 below:



(from [Mule])
Figure 4.16: Mule ESB architecture

Mule uses a compact XML-format for describing the configuration and supports Spring for lightweight specification of the desired integration. Development support is provided via Ant and Maven-scripts, there is also an Eclipse-plugin (MuleIDE) for basic configuration and deployment from within the IDE. Recently, the MuleIDE has been extended to provide a visual editor for designing integration maps, but this has not yet been ported to the new Mule 2.0 release.

Compared to a full JBI solution, Mule is easier to get started with and provides a more lightweight, configuration-centric integration model with little restrictions on architecture or message format. Components may be simple Java objects or Spring Beans, which can be directly integrated without code changes by writing a suitable configuration. This simple approach allows to quickly set up a prototypes and reduces time-to-market for small to medium size projects, but allows the solution to grow to enterprise scale when needed, because of the clustering support in Mule. Especially in small or performance-critical settings, Mule provides an advantage by transforming messages only when needed at the last possible moment, which is sometimes compared to *late binding* in programming languages. Mule has recently been integrated into an integration stack with various components working together, called Mule Galaxy[36]. Integration into existing environments is possible in several ways, either as a component inside a JEE application server or by connecting to an existing message bus over JMS or a similar protocol supported by Mule.

The deviation from open APIs, common messaging formats and core ESB principles also has significant disadvantages and bears the danger of vendor lock-in (Mule is the only implementation available) and “spaghetti-integration”, as components are still tightly coupled and statically configured, which can result in highly complex configurations as every possible message path has to be explicitly designed, catering for required translations and transports. In this way, Mule really only realizes *component* integration and not *service-oriented* integration, which would require the consistent usage of standards-based interfaces, such as WSDL, which is utilized in JBI. Also, because endpoints have to explicitly specified, there is no real location transparency, which is a core prin-

[36] http://mulesource.com/products/galaxy_features.php

ciple behind SOA. Moreover, Mule does not support hot deployment or reconfiguration on-the-fly. This has been a long standing point of criticism and will be addressed in an upcoming version by using OSGi as the underlying component model.

Much of Mule's unique advantages proclaimed in [Mule] (section *What Is Mule?*), can be also realized with JBI in a standards-compliant manner, without incurring the disadvantages of Mule, as shown below:

- *any component type is supported:* In JBI, components can be easily wrapped into ServiceEngines or Binding-Components. ServiceMix also supports POJOs (Plain Old Java Objects) directly through API extensions and provides a lightweight configuration model using XBeans (JBI 2.0 will also support OSGi components).
- *better component reuse and less API overhead:* JBI components can be reused across different runtimes, which is out of scope for Mule. This comes at the price of added API complexity, which is why many JBI implementations introduce more lightweight API-layers that allow easier development of JBI components but are not always fully JBI compatible, restricting reuse across JBI containers. JBI 2.0 will however refactor the API where it is overly complex or restrictive, and make the specification more developer-friendly in general. Recoding is also not necessary in JBI, as ServiceUnits provide rich ways of reconfiguration, even *at runtime*.
- *no restrictions on the message format:* JBI requires transformation into a normalized (but *not canonical*, see Section 4.2.1) XML-format on purpose: this reduces complexity of the integration solution and minimizes the risk of tight coupling and point-to-point integration, which can easily happen with Mule. Implementations are free to optimize message flow where possible, so transformation is not performed when communicating endpoints use the same protocol. Also, binary formats and proprietary extensions are supported through attachments and metadata, respectively.
- *support for various topologies besides ESB:* The JBI specification does not restrict implementations to an ESB topology, but allows the implementation to chose any topology that makes sense in the target environment, which has been already shown in Section 4.2.1 and previously in Chapter 3, Figure 3.10. The next version of OpenESB will support *peer-to-peer* messaging using JXTA, and other JBI implementations support various messaging and clustering forms such as JMS, SEDA or Proxying.

The disadvantages identified above and the restrictions imposed by the proprietary Mule API make it unsuitable for sustainable tool integration, esp. when comparing against the requirements and alternatives outlined before. Also, Mule 1.x has been previously tried for a ToolNet redesign in a case study, but was found to be too restrictive with regard to endpoint addresses, as it did not support dynamic endpoints (e.g., when tools come online or go offline). This limitation to static configuration impedes dynamic tool reconfiguration and agile workflows.

Mule does not fully apply best practices in SOI and enterprise integration, which may result in complex solutions that are easy to build initially but hard to maintain later. However the straightforward approach to integration, easy configurability (maximizing reuse of existing service components) and low API overhead (maximizing performance and easing development) are key advantages that should be followed in JBI 2.0 where possible.

4.4.2.4. Comparison Matrix

Table 4.3 summarizes the features identified in open source JBI implementations considered and also includes Mule for reference:

Feature \ Solution	Mule	ServiceMix	OpenESB	PEtALS
open source	yes			
open standard	no (proprietary API)	yes (JBI)		
common data format	no	yes (JBI normalized messages)		

Feature \ Solution	Mule	ServiceMix	OpenESB	PEtALS
adherence to SOA/ SOI principles	partially	fully (WSDL interfaces and XML messaging)		
API complexity	low, simple component model, allows easy reuse of existing service objects	complex specification, working with the bare API requires thorough understanding of several concepts, components have to follow the API paradigms		
Configuration	straightforward, allows reconfiguration without coding, Spring support	more complex, requires editor support, reconfiguration possible (even at runtime); ServiceMix offers lightweight configuration using XBeans		
Dynamic	high design time dynamic (components can be easily added), no runtime dynamic (no hot deployment)	medium design time dynamic (many correlated configurations, more complex API), high runtime dynamic (full support for hot deployment)		
Scalability	high, provided out of the box	medium, supported through JMS	medium, optional	high, designed for enterprise scalability
Integration in existing environments	excellent (standalone, various JEE app servers and message queues)	very good (standalone, JEE app server (Geronimo) or message queue)	good (JEE app server (JBoss) and messagequeue)	good (standalone, JEE app server (JOnAS) and messagequeue)
Components Availability	medium, many third party and still in development	very good, many components available in the JBI ecosystem, many provided out of the box, many additional components in development		
Developer Support (Docs, Community)	good (Wiki with some information, but documentation requires free registration)	very good, Wiki and forums, ChainBuilder provides full manuals	good, many examples in the OpenESB wiki	medium, manual available but incomplete, requires free registration
Community Adoption	high (also commercial)	high (commercial through FUSE product)	low, but gaining traction through tool support, see below	medium (mainly used in some European institutions and research projects, but gaining recognition)
Tool support	medium (only an Eclipse plugin provided, in development)	medium (relies on increasing JBI support in emerging Eclipse STP), but ChainBuilder adds an excellent IDE to ServiceMix	very good (full support for designing, configuring and deploying JBI ServiceAssemblies)	medium (Eclipse plugin for designing and deploying JBI ServiceAssemblies, only just released, not as powerful as NetBeans)
Maturity	Very high and proven, used in several commercial	high (stable version 3.2, version 4 in development)	high (used in commercial variant, Glassfish ESB)	high (used in widescale industry and research settings)

Feature \ Solution	Mule	ServiceMix	OpenESB	PEtALS
	settings (e.g., Wal Mart)			

Table 4.3: Comparison of Open Source JBI Solutions

JBI implementations show similar advantages but differ in tooling and developer support (API, community, documentation) and offer unique advantages in certain target environments. Mule is a very mature and reliable product, being on the market for the longest time and deployed in large banks, but limited to a single vendor in terms of support, innovation and development capacity. The ChainBuilderIDE, based on Eclipse and ServiceMix, currently provides the most capable platform for developing standards-compliant JBI assemblies, and was chosen for the prototype realization in Chapter 6.

4.5. Summary and Conclusion

JBI today is already a solid standard to support interoperable, enterprise-capable, and practical integration solutions. The use of WSDL for service description, XML for message payload and the JBI specification itself promote the standardization of state-of-the-art integration even when Java is not the language of choice of the applications to be integrated.
--from the keynote to Jazoon'07[37]

Kristen Puckett, marketing director at Bostech (Chainforge) concluded in his blog[36]: “You are working with interchangeable, vendor-independent building components that plug-in natively without special integration.”. As shown in this chapter, an open, cross-vendor standard combined with an open source implementation is pivotal for successful tool integration. By fully embracing the concepts and best practices in service-oriented integration and also supporting event-driven integration, JBI provides a perfect environment for dynamic, scalable and extensible tool integration, including the integration of legacy or COTS applications, while still facilitating reuse among components through a loosely coupled design model that cleanly separates individual integration layers.

The following chapter will introduce a case study of an existing tool integration framework, ToolNet, which will then be rearchitected by applying the findings of this chapter in the course of a prototype based on JBI.

[37] http://jazoon.com/en/conference/presentationdetails.html
[36] see the article Why choose a JBI-compliant ESB? [http://chainforge.net/pLog/index.php?op=ViewArticle&articleId=5&blogId=2]

Part II. Practical Integration: Redesigning the ToolNet Framework

Table of Contents

The practical part presents a concrete integration framework as a case study and performs an evaluation based on the findings in the proposed solution, which is then applied in the redesign of ToolNet.

A prototype demonstrates the viability of the new concepts proposed, which are carefully evaluated for problems solved and remaining challenges.

Chapter 5. Case Study: The ToolNet Framework

The system development process is characterised increasingly by its heterogeneous tool landscape. In the case of this mixture from commercial tools, legacy systems and, in-house tools a completely integrated data view can not be guaranteed. By a common backbone like ToolNet, to which all necessary tools are connected, such a fully integrated data view can be established.
--The Integration Framework ToolNet - Vision, Architecture and related Approaches

5.1. Introduction

The ToolNet-framework [Altheide2003] is a constantly evolving prototype solution that has served as the topic for several theses already, such as [Doerfel2002], [Beyer2005] and [Walter2006], which describe various parts of the framework and several refactorings of the system's architecture. This chapter therefore focuses on ToolNet's key concepts and features, in order to generate a general understanding of the framework's purpose, aims and functionality, and refers to relevant literature for further details on aspects which are not covered in this work. Then, the current architecture and implementation is analyzed, followed by a critique which examines drawbacks and limitations of the current solution. The prototype described in Chapter 6 will then try to provide solutions to the problems identified in this chapter.

5.2. Overview

ToolNet [Altheide2003] is a custom integration framework with service-oriented concepts developed by EADS Corporate Research Centre Germany to overcome the problem of isolated tools and heterogeneous data models commonly encountered in the aeronautic engineering domain: In this field, various prepackaged software applications (mostly COTS tools) are used for autonomous but closely related engineering-tasks in product development. Ideally, these tools could be connected as needed to form an integrated tool chain where data and functionality is shared among tools related to a certain task, allowing visualization and manipulation of common data elements across individual tools in a service-oriented way. In reality, these tools are developed independently, mostly by third parties, and thus are not aware of each other. This creates a discrepancy between the way tools are designed and distributed, and the way end users actually use these tools to suit their needs or meet certain requirements such as company guidelines or development processes.

There is no easy solution to this problem, as conventional approaches cannot be applied here. One option would be to adapt the tools to suit the needs above by modifying the source code. For COTS Tools, access to the source code is usually not available, but even if it was, considerable development efforts would be required to adapt such complex tools as used in aeronautic engineering, outweighing the advantages of tool integration. This also rules out replacing the tools entirely with custom implementations that offer a rich API, or building “integration-aware” components that are plugged into a central integration core and use a common design based on a shared data model, such as IPSE (Integrated Process Support Environment, c.f. [Mitschke2005:6-7]).

The solution chosen by EADS was to develop a loosely coupled integration framework, *ToolNet*, that acts as a mediator between existing, disparate applications, providing the missing functional and data integration within a service-oriented architecture. This is realized by Adapters (see Section 5.3.7) that wrap the original tools' isolated functionality inside common services and make them available to all participants of the ToolNet framework. Figure 5.1 illustrates the logical design of the ToolNet framework:[1]

[1] all figures in this chapter, except for the DOORS screenshots, were taken from ToolNet documentation and presentations kindly provided by EADS CRC Germany

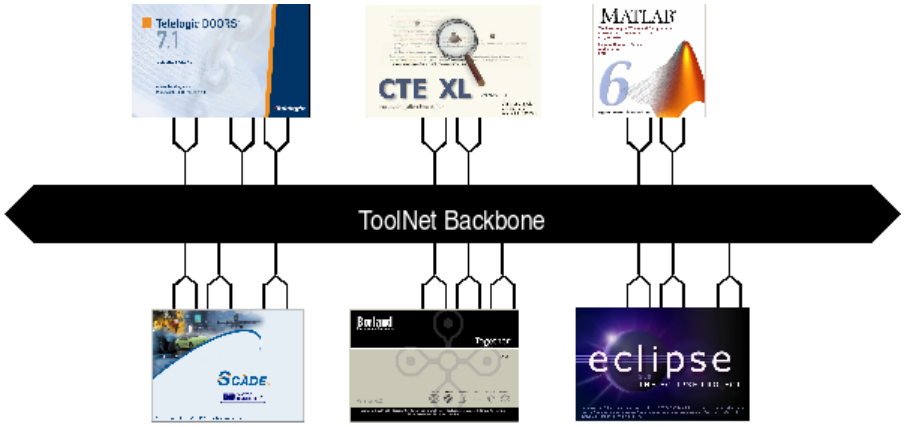


Figure 5.1: ToolNet Conceptual Overview

The framework enables users to link together existing software tools using a graphical desktop (see Section 5.3.2). This is done by manually defining the relations between the tools' data models, as detailed in Section 5.3.6. These relations are stored in a database, which also holds information about Projects (see Section 5.3.4 and Sessions (see Section 5.3.3). Tools can run on any platform and communicate with the ToolNet backbone via web services, whereas ToolNet-components use RPC for intra-framework communication (see Section 5.3.1).

As usage of cross-platform technologies and open standards is a primary requirement for integration-frameworks, ToolNet is realized in Java and tries to embrace existing standards and solutions where applicable: web services are used for communicating with tools, and the previously proprietary Adapter architecture and the ToolNet Desktop have been refactored to utilize the *Eclipse Rich Client Platform (RCP)* which is based on the open plugin-standard OSGi (see Section 3.2.4.2). Currently, the remaining ToolNet components are being refactored as Eclipse-plugins to migrate from the custom ToolNet-backbone to a fully plugin-based, standardized architecture.

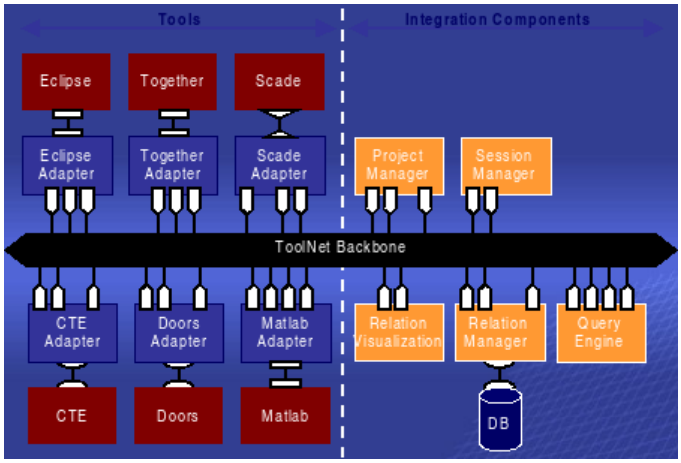


Figure 5.2: ToolNet architectural overview

5.2.1. ToolNet Challenges

An integration framework like ToolNet that is targeted at COTS tools has to work around various limitations imposed by closed tools, as mentioned in Chapter 2. Challenges include restricted interaction with tools through vendor-specific APIs (if they are available at all), proprietary communication mechanisms and data formats (this includes storage formats as well as in-memory data), and language barriers: tools may be realized in .NET or scripting languages like Python or proprietary dialects like DXL (in the case of DOORS, see Section 5.4), or

even as browser-based applications. From a user's perspective, the tools should be integrated transparently: the framework should provide access to other services from within the original tool, and allow seamless editing of common data throughout the entire tool chain, using tools as required for particular tasks or processes. Also, performance of neither the individual tools, nor the integration framework, should degrade when more tools are added. Lastly, the framework should allow users to cooperate in a workflow over the network, using tools and their services in a distributed manner. This neccessitates a scalable, dynamic and distributed framework which has find a balance between the requirements outlined before, working around the challenges mentioned without sacrificing the goal of the solution.

An important distinction to other integration scenarios, especially from the enterprise domain and backend integration, is that with a desktop integration-approach like ToolNet, existing tools are not fused into a new meta tool that virtually replaces all individual tools integrated. Instead, users continue to work with existing tools they are used to and gain additional functionality by being able to relate the tools in such a way that an integrated workflow is possible, without having to care for data exchange or having to duplicate information in several tools. The original tools stay autonomous but they can be extended by the means of Adapters that expose the tools' API as a set of services within the ToolNet framework.

Integration at the user interface-level also raises special demands regarding responsiveness, usability and transparency (for seamless integration), blending into the original tool so that additional functionality available through the integration framework appears to the user as a natural extension to the tool at hand, and not as a separate, external application that has to be used *in addition* to the original tool (which would complicate the workflow instead of making it easier and more integrated). This is radically different to integration in enterprise applications, where the user is commonly exposed to an interface-façade (e.g., a web application) that presents a unified view on a business domain-level, and directs user requests to several, disparate backend systems, combining the results and sending them back to the user in combined and revised form.

5.2.2. Terminology

The following definitions are based on [Mauritz2005] and provide some clarification on common terms which are used in a special context in ToolNet (esp., Tools vs. Components, Services), and also give some insight into the key aspects of the framework's architecture and functionality:

Tool	a software application used in the target domain (engineering) which is required as part of a model-based product development process. Usually, tools are well-known *COTS*-products, such as the requirements management tool Telelogic *DOORS* (see Section 2.2.1 for a general definition)
Adapter	a tool-specific wrapper to convert requests transferred via the ToolNet backbone into a tool's proprietary interface. An Adapter may also use services from other *tools* or ToolNet *components*, thereby extending the integrated tool's functionality.
TeT (ToolNet-enabled-Tool)	The combination of a tool and a corresponding *Adapter* that together enable the integration of a *tool* in the ToolNet environment.
Component	in ToolNet, a *component* is a self-contained functional unit that provides services to the outside environment
Service	the interface of a *TeT* or *component*, describing the operations provided. ToolNet defines 10 common *Services* that can be implemented by any framework component or *TeT*.

5.3. Architecture

ToolNet's system architecture provides a well-considered and up-to-date foundation that facilitates an extensible and flexible framework for integrating decoupled and isolated tools in a distributed product development environment.
--The Integration Framework ToolNet - Vision, Architecture and related Approaches

ToolNet was originally based on a proprietary, monolithic architecture that is being migrated to a standards-based component architecture based on Eclipse RCP, which is described in [Walter2006]. By using OSGi (specifically, Eclipse Equinox, see Section 3.2.4.2) as the underlying component architecture, ToolNet represents an adaptive and modular framework that is open to changing (integration) needs, utilizing a plugin-based, standards-based platform that has proven itself in a variety of application domains (see Section 3.2.3 for an introduction to OSGi and other component integration frameworks).

The core of the ToolNet architecture is represented by a communication backend, the *ToolNet backbone*, that connects the framework's components (like the `Relation-`, `Project-` and `SessionManager` explained below), providing communication inside ToolNet, and also integrates external applications through `Adapters` (see Section 5.3.7) that expose the Tool's functionality as `services`. This allows ToolNet to use services provided by external tools but also vice versa, giving tools access to services provided by other tools. This flexible approach resembles a service-oriented architecture with loosely-coupled components, overcoming the challenges and limitations outlined in Section 5.1.

Adapters not only provide a functional integration by exposing as much of the Tool's functionality as possible, but also realize data integration by exposing the Tool's data model to other tools connected to the ToolNet infrastructure. ToolNet avoids the complexity of model-driven integration and the inherent danger of inconsistencies when building metamodels from data models that are likely to change (as is the case with most COTS tools when upgrading to a newer version) by linking individual data elements from one tool's model to corresponding elements of another tool's model (see Figure 5.3). As model elements are only referenced, not copied, the original tool retains control over the data and there is no need for involving advanced synchronization or replication techniques. This approach also facilitates an incremental tool integration, allowing users to gradually evolve the integration as circumstances (e.g., time, effort or the Tool's API) permit.

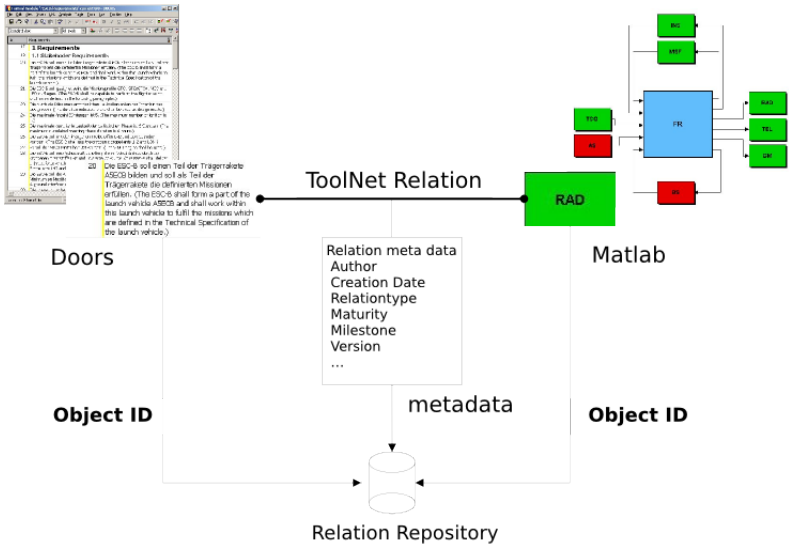


Figure 5.3: Tool and Model Relations in ToolNet

An ideal integration solution would silently work behind the scenes, connecting tools as needed and allowing the user to seamlessly work from within the original tools, editing related data and invoking functions of interest,

regardless of what tool actually provides that function. In reality, there is still a need to interact with the integration framework, such as performing management tasks or controlling the lifecycle of individual components. Also, aspects that cannot be integrated into the tools themselves have to be exposed through other means in order to provide the needed integration functionality. For example, when a tool does not allow extension of its user interface in a suitable manner, the data linking functionality needs to be presented in an alternate way so that the user can still create relations between the tool's data models.

To fill this gap, the ToolNet Desktop (see Section 5.3.2) was created to provide a central user interface for interacting with the integration framework itself and also with integrated Tools (TeTs). It is also implemented as an RCP application to follow the Eclipse-based architecture of the remaining framework. The desktop is covered in more detail in Section 5.3.2.

Finally, a database (PostgreSQL) is used for storing configuration data and for caching Objects and Relations created by users. Relations are not persisted and have to be recreated in subsequent ToolNet sessions.

The remainder of this section provides an insight into the individual ToolNet components and how they work together. Section 5.4 provides a sample use case that shows the current approach with integrating the COTS tool DOORS. This use case is later used as a reference in the prototype implementation in Chapter 6.

5.3.1. ToolNet Backbone

The framework's core is a server component implemented as an Eclipse RCP application that bootstraps the framework by setting up the other ToolNet components (originally implemented as custom components, which are now being refactored to OSGi bundles). The backbone implements a general, *SOAP*-based communication layer currently realized with the now proprietary web services-framework GLUE, enabling a distributed collaboration among tools and users. Other ToolNet components communicate over the backbone by sending SOAP-messages, using it as a *Channel* to transmit service requests and receive responses. Adapters could technically bypass the backbone and call other Adapter's services directly, but for security reasons this is usually discouraged in favor of using a Proxy (`Mediator`) that propagates service requests to remote targets.

The backbone is also responsible for database access, providing a common storage interface which is abstracted from the underlying PostgreSQL-database with Hibernate. Lastly, it provides infrastructure services like component management, keeping a Context of active Components and their related Sessions and Projects, as well as caching which is used for speeding up ToolNet Services such as the Preview-Service (see Section 5.3.5). Figure 5.4 provides an overview of the ToolNet backbone.

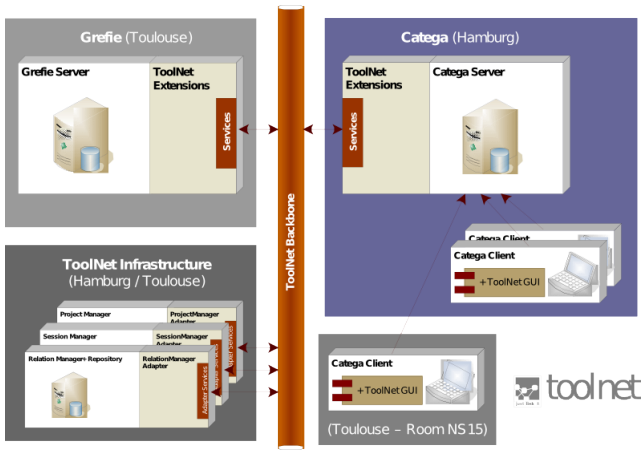


Figure 5.4: ToolNet Backbone with distributed clients (overview)

When the ToolNet server is started, it discovers available Adapters, which then register the provided functionality with the SessionManager (see Section 5.3.3) as common ToolNet Services (see Section 5.3.5). Following the classification introduced in Chapter 2, the backbone constitutes the functional integration layer by integrating the Tool's Services as provided by the ToolNet Adapters.

5.3.2. The ToolNet Desktop

The ToolNet Desktop (shown in Figure 5.5) is also implemented as an Eclipse RCP application and acts as a graphical user interface to ToolNet. The primary function of the ToolNet Desktop is to provide a user interface for Adapters so that users can discover and invoke services from available TeTs (e.g., requesting a preview from a tool's data element). This way, limited integration possibilities in the original tool, where the addition of custom menus or functions may not be supported, can be overcome and a unified user interface can be provided as an alternative, embedded into the ToolNet Desktop.

The desktop is divided into Views (Eclipse SWT GUI components) which show all active Sessions and the containing Services. An error log is provided for diagnostics and shows system events such as starting and stopping of Services and related errors. Through the plugin-based architecture of RCP-applications, the ToolNet Desktop can be extended with additional functionality by adding plugins (in the same way as ToolNet itself can be extended by adding new plug-ins such as Adapters). This mechanism is used in Adapters to expose tool-specific functions inside a common user interface.

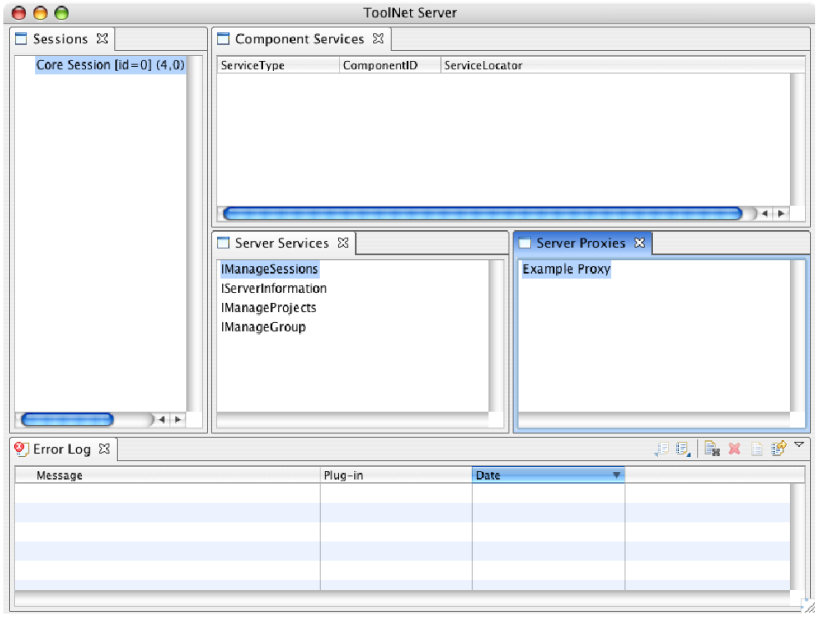


Figure 5.5: The ToolNet Desktop

The interface also provides access to common Services (see Section 5.3.5) such as the RelationService, allowing users to create and navigate Relations between tools' data models (see Section 5.3.6), which are visualized in different forms (e.g., as a tree or graph). Data elements can be previewed or highlighted in the corresponding tool, Projects (see Section 5.3.4) can be loaded or created, and `InteractionSessions` can be initiated for collaboration between ToolNet users.

Basic lifecycle management is provided, allowing Adapters and Tools to be started or stopped, as well as basic system maintenance functions such as shutting down the ToolNet server. In summary, the ToolNet Desktop can be seen as a tool dashboard, integrating the individual tools' data views in a centralized, composite user interface,

thereby providing integration at the presentational level (see Section 2.3.5). At the same time, the Desktop acts as a management console by providing functions for system administration and maintenance.

5.3.3. Sessions

The SessionManager (c.f. [Walter2006:16]) acts as a service-repository where Adapters register Services they provide, wrapping the functionality of the integrated tool and mapping it to one or more of the common ToolNet Services defined in Section 5.3.5 below. More precisely, the SessionManager holds three Sessions that group ToolNet Services according to the context:

- *CoreSession:* this Session is a Singleton that holds the global registry containing Services that should be available to all ToolNet Components, e.g., the ObjectPreview-Service (see below).

- *ProjectSession:* Services related to individual Projects are grouped under a single ProjectSession that manages service requests specific to the active Project the user is working in. For example, requesting information about a model element will trigger all ObjectInformation-Services in the current ProjectSession.

- *InteractionSession:* this Session contains user-related Services within a ProjectSession, e.g., when creating Links between two Tool's models, a RelationCreation-Service is used within the active InteractionSession.

The SessionManager implements a central Service called ManageSessionService that provides access to a global ToolNet-repository, which realizes a common abstraction from Services provided Tools (through Adapters) and other ToolNet-Components and facilitates a unified lookup and usage of Services throughout the framework.

Sessions are not persisted and have to be set up each time ToolNet is started. The CoreSession is started automatically by the ToolNet Server, whereas ProjectSessions and InteractionSessions have to be manually set up by the user for every Project she wants to collaborate in.

5.3.4. Projects

In order to meet the requirement for transparent tool integration and to attain the goal of an integrated workflow, ToolNet abstracts from individual tools by viewing tools in the context of Projects, which aligns with the project-based workflow that users commonly follow. The `ProjectManager` acts as a Service-repository for Services that manage Objects and their relations, such as the ObjectPreview-Service. These Services do not belong into the global Service repository because they are only useful in a Tool/Project-specific context. The necessary information is provided by Adapters, which register Tools and their Objects in the ProjectManager, where they are integrated as abstract data sources and connected to other data sources using Relations (see Section 5.3.6). The relation between Projects, ToolNetObjects and Links is illustrated in the UML diagram shown in Figure 5.6 below:

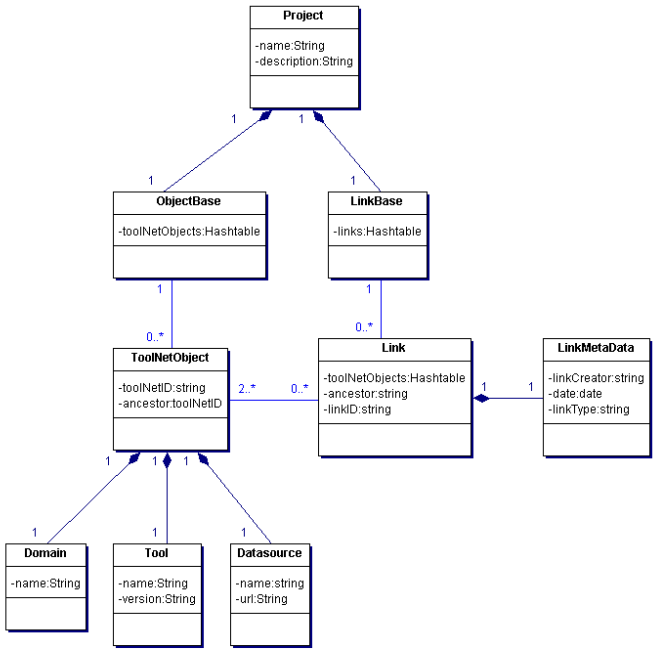


Figure 5.6: Project class diagram

The ProjectManager-component provides the necessary application-level integration (see Section 2.3.4.1) that is essential for a successful desktop integration solution.

5.3.5. Services

ToolNet defines several CoreServices that can be used globally throughout the ToolNet infrastructure (see [Doerfel2002:97-101] for an overview and appendices C, D (ibid.) for a complete list). Adapters provide an implementation for each CoreService they support by mapping proprietary tool functionality to these common ToolNet interfaces. ToolNet Services are implementations of one of the three common Service-Interfaces that map to the Session types mentioned in Section 5.3.3: CoreSessionService, ProjectSessionService and InteractionSessionService. Examples for Services include:

- *ObjectInformation-Service:*

  returns a textual description or a visual preview of a linked data element, which is passed to the ToolNet Desktop as XML with an associated XSLT, wrapped in a SOAP-message

- *Presentation-Service:*

  provides the ability to highlight individual data elements in a related Tool, e.g., the user could select a requirement in the ToolNet Desktop and request the PresentationService to highlight the element in the DOORS application

- *RelationCreation-Service:*

  This is a fundamental Service that allows users to create a Link between two corresponding data elements from separate models, each owned by a different tool (see Section 5.3.6).

ToolNet Adapters are implemented as OSGi plugins and can thus be managed as modular components from within the ToolNet desktop, resulting in a dynamic system where Adapters can be started and stopped as needed.

The dynamic installation of new Adapters or uninstalling unneeded Adapters is currently not supported by the framework.

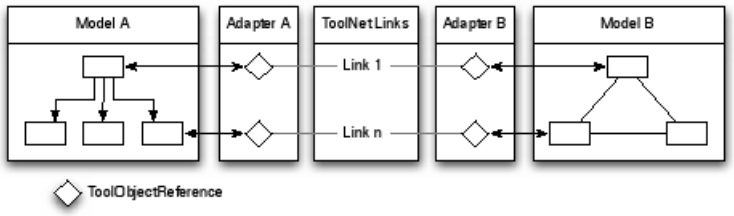
By defining new Service types, the functionality and usefulness of the ToolNet framework can be extended and adapted to new requirements. Adapters that want to provide these new Services have to implement the new ServiceInterfaces, and for CoreServices, the ToolNet Desktop has to be extended in order to provide an interface for the additional functionality. In the same way, Adapters have to be adjusted when a tool is upgraded and new features are to be integrated, or the API has changed and previously integrated functions cannot be accessed with the existing Adapter interface.

The concept of generic ServiceTypes that are transparently realized by Adapters provides a common, serivce-based interface across the ToolNet-infrastructure. This allows for dynamic queries that allow users to access Services in a transparent way. Instead of addressing concrete endpoint URIs, it is possible to specify the desired functionality or Service, and the ToolNet-framework transparently queries all available Adapters that implement the requested Service. Results are then sent back to the user over the ToolNet-desktop or they are received by a Tool Adapter which displays them in a form suitable for the tool it integrates. The search scope is specified through the hierarchical organization of ToolNet-Sessions and Projects, so only related Adapters are queried.

5.3.6. Relations

Relations are the cornerstone of the ToolNet integration framework, bridging the gaps that isolate autonomous tools and impede a project-centric, dynamic workflow. This is also reflected in [Beyer2005], where it is made clear that ToolNet's “primary aim is to integrate autonomous tools […] by offering a logical linking between their data-models.”.

As already mentioned, an important distinction that sets ToolNet apart from model-based integration-approaches is that ToolNet does not construct a meta-model that combines all data models of integrated tools, but only references selected model elements using abstract `ToolObjectReferences`, which are used as a common data description in ToolNet. Through the use of a `RelationCreationService`, users are able to link model elements from one tool to related elements in another, which is illustrated in Figure 5.7 below:



(source: [Walter2006])
Figure 5.7: Linking Models in ToolNet

The mapping from abstract `ToolObjectReferences` to the Tool's proprietary data representation is realized by the already mentioned Adapters in cooperation with a core ToolNet Service, the `IDMapper`, which are both described below. When a data element is changed in one tool, a `ChangeEvent` is fired by an infrastructure service, the EventService, to inform other Components of the modification and subsequently update corresponding data elements in related tools. Events are transmitted to all registered Components. A fine grained change propagation is not supported, also synchronization is not provided.

[Mitschke2005:13-16] shows some examples and illustrations of using Relations with the example of the DOORS application, which is described in Section 5.4 below.

5.3.7. Adapters

Adapters provide a service-oriented interface to prepackaged tools by exposing the Tool's functionality as completely as possible to the ToolNet infrastructure and consequently to other Adapters. Through this functional integration, Adapters hide proprietary communication mechanisms necessary to interact with external tools, like vendor-specific APIs realized as C, C++, Java or scripting-interfaces. More enterprise-oriented products, such as SAP, increasingly offer web service-based interfaces, which are easier to access and integrate since they are already designed in a service-oriented fashion. As a consequence, for each tool and API, there is a separate Adapter-implementation available in ToolNet that integrates as good as possible with the target technology, e.g., in Java, C# or a (mostly tool-specific) scripting language such as DXL, which is provided by Telelogic DOORS, or M-Script for Matlab, but also through web services where applicable, as illustrated by the example of the ToolNet WSDL in Figure 5.8 below.

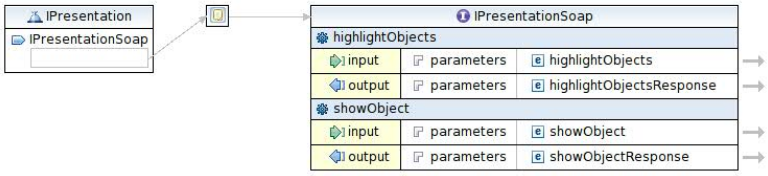


Figure 5.8: ToolNet WSDL for integration using Web services

Besides exposing the tools' functionality as common ToolNet Services, Adapters also have to integrate the tool-specific data model by mapping individual model elements to common ToolNet ObjectReferences. This mapping, which realizes the actual data integration, is the foundation of the Relation-Service introduced above, and is managed centrally by a framework component, the `IDMapper`. This component holds a lookup-table that translates between common references (`ToolNetIDs`) and tool-specific data elements, which are accessible to all registered and active Adapters. By providing a mapping from `ToolNetID`s to individual data elements in integrated tools, Adapters make them accessible to other Adapters that perform an analogous mapping to a related data element in another tool. This mapping also works for 1:n and n:m-relations.

Data mapping and integration at the user level through Adapters are the *key features* of ToolNet. Figure 5.9 shows a conceptual view of the Adapter architecture and its relation to the Toolnet backbone:

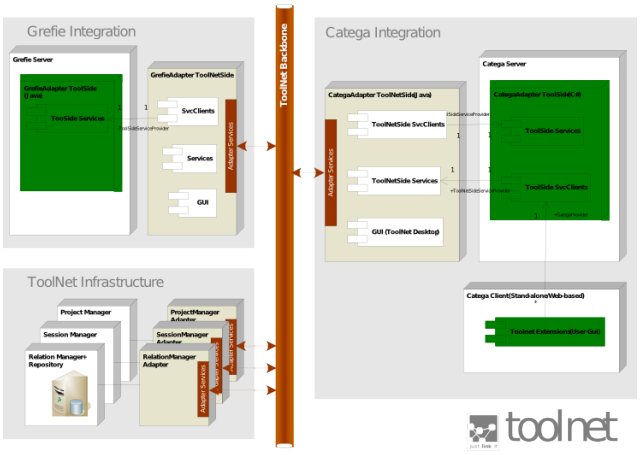


Figure 5.9: Adapters connected to the ToolNet Backbone

Adapters face the challenge of *divergent requirements*: On the one hand, they should abstract from individual tools and provide their functionality as common services inside the integration solution, but on the other hand,

they should allow full access to tool-specific functionality and data. This design dilemma is solved by dividing the Adapter in two parts: one part, the *ToolSideAdapter*, is connected to the tool, the other part, the *ToolNet-SideAdapter*, is connected to the ToolNet backbone. This separation is also necessary from a technical aspect, because integrating a tool could require an Adapter to be implemented in a different language (e.g., .NET) than ToolNet (Java).

The `ToolSideAdapter` is responsible for the functional integration and can be realized as best suits the tool, e.g., as a C++ plugin or even as a script. Some tools allow direct integration into the user interface, such as custom menus or buttons, while others only provide a limited API that has to be accessed from outside the tool.

The `ToolNetSideAdapter` is realized as a ToolNet Component that runs inside the ToolNet environment. It translates the functions exposed by the ToolSideAdapter into Services that can be used by other ToolNet components. Conversely, it may also use Services provided by other Adapters, thereby extending the functionality originally provided by the tool. This part of the Adapter is usually responsible for data integration, mapping from tool-specific data elements to common ToolNet-Objects by implementing the RelationService.

Through `ToolLinks`, the two Adapter-parts can communicate with each other using techniques such as *inter-process-communication* (for local applications), distributed communication over sockets or RMI (for integrating backend tools), or simply shared files (see also [ESB] for related integration strategies), and thus provide a transparent but at the same time highly flexible and adaptable solution to users.

Adapters usually provide a part of their user interface inside the external tool (as the tool's API allows), and the other part is realized as a plugin inside the ToolNet desktop. The user interface allows for starting and stopping tools and for accessing tool-specific functionality. The ToolNet API does not mandate a specific organization of Adapters in this respect: in case of backend tools, where usually no user interface is needed, the Adapter does not have to implement a separate user interface in order to integrate the Tool; the functionality is made available to the backbone in the form of Services that can be accessed by ToolNet Services or Adapters without user-interaction. For the opposite case, where tools are open enough to allow for full customization of the user interface, the ToolSide Adapter can realize the complete user interface directly inside the original tool, without the need for a ToolNet Desktop-plugin. For most cases however, a mixed approach is usually the most feasible solution.

As can be seen, the quality and user experience of ToolNet is highly dependent on the available Adapters and their implementation. Together with Relations, they represent the foundation of the ToolNet vision. In order to adapt to the divergent needs of integration developers, ToolNet provides a flexible Adapter design that leaves enough room for tool-specific integration-approaches without sacrificing the framework's unified approach.

5.4. Case Study: Integrating DOORS

There are things known, and there are things unknown, and in between are the Doors.
--Jim Morrison, The Doors

As an example for tool integration with ToolNet, this section introduces a requirements-management tool that has successfully been integrated with ToolNet, Telelogic DOORS [DOORS], and shows how the necessary Adapter has been realized. This example will also serve as a use case for the prototype (see Chapter 6) later.

5.4.1. Introducing DOORS

DOORS (short for “Dynamic Object Oriented Requirements System”) is a widely used commercial application for requirements engineering, a field in software development that ensures a product's conformance to the customer's requirements and to relevant standards and regulations. [REH] discusses this topic thoroughly from a general perspective; a good introduction is provided together with DOORS by Telelogic, *Get it Right the First Time: Writing Better Requirements*).

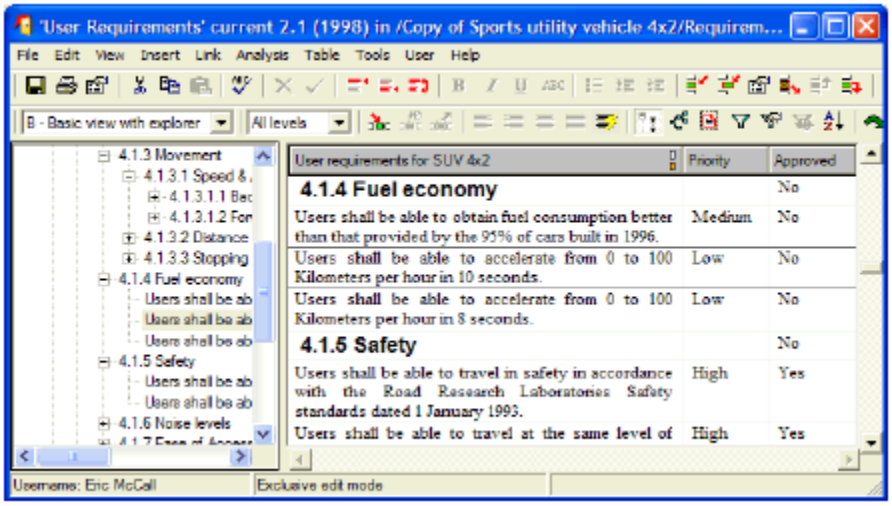


Figure 5.10: The DOORS Interface

DOORS represents individual requirements in a *Project* as a hierarchical structure of attributed Objects and Modules. *Objects* consist of a leaf node that contains a heading, and a text node with some content, which can be comprised of text, images, diagrams or embedded documents linked to an external application (using *OLE* or ActiveX). Related Objects are grouped into *Modules*, which hold information common to all contained Objects. A Project acts as a container for all modules and also implements user rights management and other administrative tasks. Using system-defined and custom Attributes, Objects and Modules can be annotated and typed, e.g., with a Priority or Approval-status, as shown in Figure 5.10. Attributes can be later used for filtering, which helps organize Projects in task-oriented Views.

A key feature of DOORS is *linking*, which is illustrated in Figure 5.11: Related requirements can be connected by linking together Objects. This way, requirement interdependencies and hierarchies can be expressed and the user is supported in managing the resulting requirement networks which can become very complex for large projects, as common in the aeronautic domain. Changes in requirements can be traced so that dependent Objects can be identified and updated, which also allows the user to better understand the impact resulting from that change. This ensures consistency of requirements throughout large, dynamic long-term projects that are likely change over time.

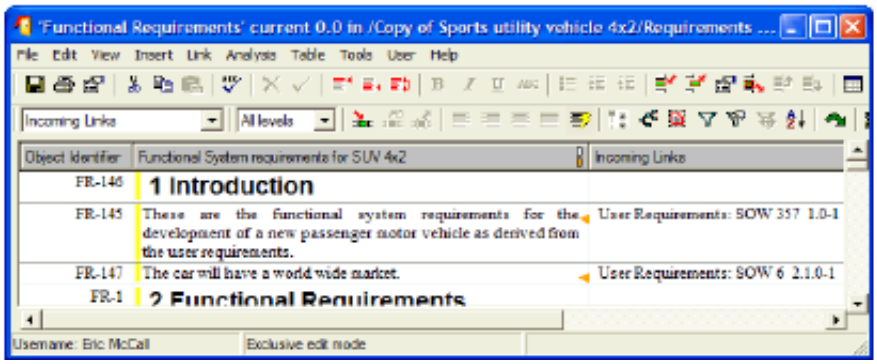


Figure 5.11: Linking Objects in DOORS

Another important feature of DOORS is rich support for importing and exporting various document formats, such as extracting requirements from plain text, Word documents, spreadsheets (in CSV, TSV or XLS format) or project management tools like Microsoft Project. Once imported, requirements can be edited, annotated and linked in DOORS, complemented with additional information (textual or graphical), and finally exported into a supported format.

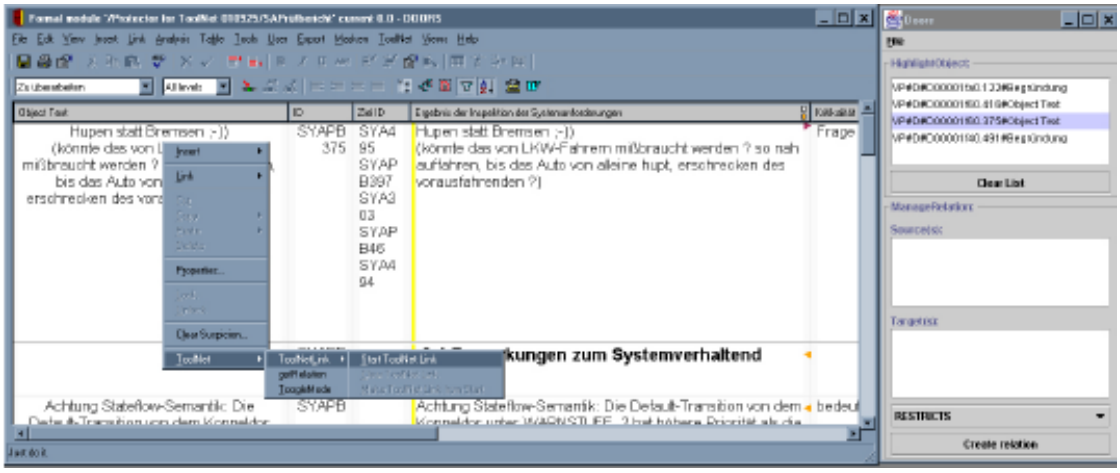
The Import- and export-functionality is realized through a powerful scripting interface, which is very important for integration into ToolNet: *DXL* (DOORS eXtension Language), which is covered in detail in [DXL] and defined as “an easy-to-learn scripting language that you can use to control and extend DOORS functionality.” (see [DOORS], *Using DXL*). The C/C++-like language exposes the functionality of DOORS to external applications and provides a comprehensive API for controlling most aspects of the DOORS interface, allowing for manipulation of Objects, Modules, Projects, as well as any associated Attributes or Links.

In addition to the scripting interface, a C-library is provided for socket-based inter-process communication (*IPC*). This interface can be used by other applications that wish to interact with DOORS by sending DXL-commands over a `TCP/IP` socket-connection or a `UNIX` pipe. This library is used in ToolNet's DOORS-Adapter (see below) as well as in the prototype (see Section 6.4.2).

More information about DOORS and its API is available in [DOORS].

5.4.2. Integrating DOORS: The DOORS Adapter

With the DXL scripting interface, DOORS is well suited for integration with other software tools, such as Rational Rose, which has been demonstrated by IBM in [PLUSS]. In this project, DOORS was integrated with the software modeling tool Rational Rose using DXL to provide a combined tool chain for the PLUSS use case modeling approach. For ToolNet, a more general solution was needed to utilize the framework's service oriented integration-approach, which required the realization of a ToolAdapter that would utilize the DXL scripting-interface for controlling DOORS and the necessary `TCP/IP`-based IPC, which is provided by a C-library included with DOORS. [Doerfel2002:G] covers analysis, design and implementation of a ToolNet-prototype which integrates DOORS on one end over the ToolNet backbone with MatLab on the other end, using application-specific scripting interfaces (DXL for DOORS and M-Script for MatLab) for accessing the tools' functionality and user interfaces. For integrating ToolNet with DOORS, a *JNI*-based Java-wrapper (see Section 3.2.2.2 for an introduction to JNI) was implemented to access the C-library provided by DOORS, and DXL-scripts were realized to extend the DOORS user interface with custom menus and functionality. For example, a ToolNet window was added where users can easily access ToolNet-provided functionality like inspecting and linking Objects. The result of the integration is shown in Figure 5.12 below.



(from [Doerfel2002:133])
Figure 5.12: Creating a ToolNet Link from within DOORS

In detail, the previously mentioned DOORS C library provides a higher level interface to the `TCP/IP`-based communication with DOORS through the functions `apisend()` and `apireceive()`. Although direct communication using sockets is possible and has been tested (as shown in Section 6.4.3.5.3), using the library functions has additional benefits: the socket communication is handled by the library and possible errors are captured

and transparently reported to the application as status codes. Also, callback functions for errors and other events can be registered, even the definition of custom DXL-like scripting languages is possible in order to facilitate a seamless integration with other tools and environments. The DOORS C API is described in [DOORSAPI:7-15], integration is covered in [DOORSAPI:c5].

The ToolNet DOORS Adapter's IPC connection to DOORS is realized with *JNI* in a separate wrapper class, the `DoorsCAPIWrapper`. This wrapper is written in Java and realizes a Java native interface that contains stub functions implemented by a custom C library (`DoorsCAPIWrapper.dll`), which acts as a bridge to the DOORS C library (`dxlapi.dll`). Outgoing communication with DOORS is done by calling the library's `apiSend`-function (using the native wrapper), listening for incoming DOORS commands is implemented as a normal `Socket.listen()` operation in Java, as there is no matching C API function for receiving input from the IPC channel.

The existing DOORS Adapter was also taken as a starting point for the prototype implementation of the proposed solution, which is covered in Chapter 6, although a different approach was used for the final prototype implementation (as described in Section 6.4.1.3).

5.5. Evaluation and Critique

ToolNet has grown from a prototype, as introduced in [Altheide2003], to a mid-sized, distributed tool integration framework with Adapters for several proprietary COTS tools. This evolution has led to a grown architecture that is gradually being migrated from a proprietary hub-and-spoke architecture to the OSGi-framework and Eclipse RCP. Using Eclipse as an integration platform has been shown to be insufficient, as detailed in Section 3.2.4.2. An integration solution of this scale and flexibility needs an additional layer of abstraction, but OSGi/RCP only provide component abstraction. The Eclipse platform does not address integration of non-Eclipse or COTS components, e.g., by providing a standard Adapter design. Neither does it provide any service-oriented integration for integrating on the application level beyond functional integration. Also, higher-level integration issues like semantic data integration and advanced messaging facilities – e.g., using a common message format that allows translation, extension (through metadata), or routing – are missing. The framework is thus rather static and cannot be reconfigured at runtime, making it impossible to add, update or remove Adapters as needed. Also, tools have to be started from within the framework, which breaks integration transparency from a user's point of view.

The Adapter architecture tries to separate tool-specific Adapter functionality from general framework functionality, but the design is based on proprietary interfaces that do not provide much common functionality. This results in poor reuse, e.g., each Adapter has to implement web service communication if communicating with external Adapter endpoints. Because endpoints can be accessed directly using the target address or class name, there is no location transparency (Adapters have to decide whether to use the Proxy or directly communicate with the target endpoint). The Adapter design also does not provide a clean separation of integration layers, such as business or application logic (that reflects how a tool works), data semantics (translating the tool's data format) or protocol logic (describing how the tool can be accessed). This results in Adapters growing more complex and inflexible as tools provide or migrate to different interfaces or data formats, because all combinations have to be hard coded: several versions of a tool might be used at the same time, and dependent on the project's needs, different interfaces of a tool may have to be integrated. A separation of integration levels would allow dynamic configuration between tools as needed.

The core communication architecture still relies on a custom backbone with close dependencies on the remaining framework, including a proprietary web service stack (GLUE). The description in Section 5.3.1 closely resembles an *Enterprise Service Bus*, but there are some major differences between the custom solution encountered in ToolNet and a standards-based messaging backbone of an ESB, as defined in Section 3.3.3.2, including advanced routing, mediation, location transparency and dynamic Service discovery (only broadcasts to all Adapters that implement a specific ServiceType is supported). While basic event-driven concepts are supported, a full EDA supports more advanced concepts like complex event-processing or event streaming (see Section 3.3.5). This

functionality is already available as proven open source implementations that can be reused in existing ESB implementations, so there is no need to reimplement an EDA using a custom API.

Lastly, instead of standards-based management access, ToolNet provides a custom management UI, the ToolNet desktop, which is based on Eclipse RCP but uses a custom UDDI-like approach for querying Adapters and Services. Migrating to the JMX standard, exposing Adapters and Services as *MBeans*, would open up ToolNet to web based access and enable integration into an existing management infrastructure based on standard management-protocols like SNMP (see Section 3.2.3.2).

5.6. Conclusion

The ToolNet framework provides an interesting case study for COTS tool integration and shows several promising approaches that have been noted in literature. The vision of keeping tools decoupled from the framework and using tool-specific Adapters allows for both transparent and deep integration of tools, tightly integrating with the original tool's interface, all the way up to the user interface. This sets ToolNet apart from purely model-based solutions that concentrate on data integration but leave out interface integration, resulting in additional efforts necessary for analyzing and designing a metamodel, and for applying formal methods to generate needed Adapters (which impedes reuse of existing COTS Adapters).

ToolNet proposes a novel approach for data integration: data is kept in the original tools but connected through user-defined Relations which can be navigated in both ways. This allows users to link tools based on common data objects as needed, without having to create a complete metamodel, which is often impossible in a heterogeneous tool landscape where data models do not overlap sufficiently. Functional integration is provided by Tool Adapters, which allow users to operate on integrated tool's data across tool borders, by using existing Relations. This enables transparent workflows without manual integration work necessary to bridge data and functionality between incompatible tools.

The design and implementation of ToolNet faces several challenges and limitations, as has been noted in the previous section. The solution represents a custom Adapter-framework with high inter-dependencies, and facilitates tightly coupled hub-and-spoke integration. During the lifetime of the project, several standards and solutions from enterprise integration have become available: from Adapter-standards like JCA to service-oriented integration frameworks like WSIF and recent higher level integration solutions like SCA or JBI. Current ESB implementations provide a much richer and more advanced communication backbone based on service-oriented messaging and event-driven architecture, whereas the current, custom solution is based on a proprietary, static API. This view is also shared by [Mauritz2005], who concludes that “a similar architecture with the ToolNet's bus for constructing integration systems from plug-in components is emerging in the J2EE world through Java Business Integration”. Because some newer ToolNet Adapters already use WSDL for exposing their interfaces, a move to JBI would be feasible and provide a smooth migration path (see also Section 7.4).

The ToolNet framework needs to be refactored and retrofitted to embrace existing standards, including emerging standards like SDO for data integration and high-level integration standards that can be used for service-oriented, dynamic tool integration. A possible solution is presented in the next chapter, using *Java Business Integration* and Apache ServiceMix.

Chapter 6. Prototype ToolNet/JBI

In theory, there is no difference between theory and practice. But, in practice, there is.
--Jan L. A. van de Snepscheut

6.1. Motivation and Overview

To demonstrate the findings outlined in previous chapters, and to show a possible solution that overcomes the problems of current desktop integration approaches such as ToolNet (see Section 5.5), a prototype ("ToolNet/JBI") has been developed that applies the main concepts and technologies from enterprise integration to a concrete integration problem on the desktop. The prototype implements a subset of the new architecture (see Section 6.4), using the *Java Business Integration* (JBI)-standard for integrating the commercial requirements engineering tool Telelogic DOORS (see Section 5.4), which serves as a typical example for integration of COTS tools on the desktop.

The prototype acts as a proof of concept, showing the advantages and challenges of the proposed JBI-based architecture, as outlined in Chapter 4 and shown in Figure 4.13, with the example of an integration scenario that has been previously implemented with the ToolNet framework, which is described in Chapter 5. The prototype stays true to the original vision of transparent desktop integration over a common backbone between COTS applications, enabling users to access objects in one tool from the other, while retaining the original interface the user is accustomed to. In order to demonstrate the flexibility of the new approach and to present a possible migration path, the original ToolNet Adapter-scripts have been reused and transparently communicate with the new implementation. Also, a basic UI is available in the form of a JMX management interface (see Section 6.4.3.5.4), which mimics the core functionality of the ToolNet desktop and allows basic interaction with the Adapter and DOORS.

In order to stay within the scope of a thesis project and to allow for easily comparing the new solution to existing approaches, the prototype only implements a single Service from ToolNet, `HIGHLIGHT_OBJECT`, that is exposed over the JMX interface (which substitutes the ToolNet desktop). When the user invokes the Service, the corresponding requirement object is brought into focus in the DOORS application. These objects are connected to the prototype by invoking certain operations in the ToolNet-menu in the DOORS interface, namely Select Object as Source or Select Object as Target. This creates a relation, as defined in Section 5.3.6, that is visualized as an MBean that representing the Object in the prototype interface and providing the aforementioned `HIGHLIGHT`-operation.

The remainder of this chapter outlines the original requirements and what has been identified as out of scope for the prototype. Later, the design and implementation is analyzed and the rationale for the concrete solution is covered, which is finally put to work in a showcase that realizes the use case described above. An evaluation of the chosen approach follows in Chapter 7, which also provides a detailed comparison between the new solution and the existing ToolNet implementation.

6.2. Goals

To address the shortcomings and problems of the current ToolNet-architecture, as outlined in Section 5.5, the prototype should introduce a possible solution by addressing the problems and requirements defined below. Due to the limited scope of this thesis, the proposed solution can merely serve as a demonstration of the core concepts introduced by the new architecture, which is described in Section 6.4.2. Section 6.3 discusses limitations and missing parts of the prototype, which would be necessary for migrating the complete ToolNet-framework. The main goal of the prototype implementation is to show that the proposed architecture can be implemented with the selected technologies, and that it has potential to offer a practical solution to the limitations and problems of

the current ToolNet architecture and implementation. For a survey on general requirements in tool integration, see Section 4.1. The following sections describe the goals for the prototype in more detail.

6.2.1. True COTS Integration

First and foremost, the prototype should follow the ToolNet vision and offer a solution for integrating COTS tools, so it should not assume the availability of Java interfaces (e.g., in the form of Beans) or web services. To match this requirement, the prototype should integrate the DOORS tool (see Section 5.4 for a detailed description). Telelogic DOORS is a good test case for an integration scenario, being a closed-source, commercial off the shelf-application with a public API and a proprietary scripting interface.

6.2.2. New Service Backbone

To address the shortcomings of the existing, proprietary ToolNet backbone, an existing, standards-based, dynamic and powerful service backbone shall be evaluated. The new backbone should offer dynamic registration and installation of services, understand the notion of ServiceTypes (as described in Section 5.3.5), provide support for orchestration, management and security, as well as clustering. This requirement is best met by using an ESB implementation that supports common standards for web services and offers some integration facilities. As proposed in this thesis, JBI provides a standard for service-oriented integration and a Java API for realizing such a solution, thus a JBI-enabled ESB-implementation was chosen, as described in Section 6.4.

6.2.3. Redesign of the Adapter Architecture

The Adapter-architecture should be refactored to facilitate reuse and allow more rapid integration of new tools. The new architecture should provide Adapters with more runtime-flexibility by offering location-transparency: Adapters should be able to run on the server- or client side, whichever is more appropriate for the tool to be integrated. Adapters should be easily extensible, allowing for adding new services or adapting existing ones when new tool versions become available. This should be possible in a timely manner in order to utilize new or changed tool functionality as quickly as possible. When new *Service(Type)s* are added to ToolNet, Adapters should be able to maintain compatibility as far as possible, while at the same time a quick adoption of new ToolNet functionality should be facilitated. With the current solution, this is hard to achieve, since there is no common core functionality or API that Adapters could utilize, and as a result they are implemented rather isolated (see Section 5.5).

The aforementioned points are an important step in finding a common architecture which combines the rich and varied set of goals and integration-scenarios a framework like ToolNet has to support (see Section 5.2.1 for further discussion).

6.2.4. Support for Non-Java Languages

Tools that offer an API often target different languages than Java, either more system-level languages like C, high-level alternatives to Java like .NET, or scripting languages like Python, Perl or Ruby, or even proprietary languages like DXL in the case of DOORS, or M-Script in the case of Matlab. Although the ToolNet framework itself is realized in Java, it should be possible to implement Adapters in and for other languages, so that available APIs can be utilized without having to reimplement the API or provided libraries in Java, which would unnecessarily hinder integration of tools and complicate the development of Adapters.

6.2.5. Independent Implementation

The prototype should be realized as a standalone solution in order to provide a clean implementation of the proposed architecture, and so that it can be better compared to the current ToolNet implementation. As a result,

the prototype also serves as a conceptual base for an incremental migration at a later stage: functional or logical overlaps may be identified more easily and possibilities for connecting existing parts of ToolNet with the JBI-based architecture become apparent. This allows for the definition of a migration plan where the most needed features of the new architecture are implemented first and connect to the existing solution using a bridge (thus integrating two integration frameworks, creating a meta-integration-framework…). This aspect is covered in more detail by Section 7.4

6.3. Non-Goals

This section briefly discusses goals that were identified as beyond scope and not feasible for a prototype implementation.

In order to keep the implementation effort in scope, the prototype only implements one Adapter to integrate a single application, Telelogic DOORS. This way, some key concepts of the new architecture are applied to provide a new approach to an integration scenario that has been implemented in the current ToolNet release. This makes it easier to compare the implementations and evaluate strengths and challenges of the proposed solution.

At the same time, this means that one of ToolNet's key feature, *Relations* (detailed in Section 5.3.6), cannot be fully realized in the prototype. The implementation of these inter-tool links that connect common information in separate data models of integrated tools depends on several parts of the ToolNet infrastructure, including the IDMapper, the RelationManager, and finally the ToolNet Desktop, which is well beyond the scope of a simple prototype. On the other hand, implementing only one Adapter would limit the prototype scenario to unidirectional integration, and much of the integration and interaction possibilities would be unavailable. As a solution, the new Adapters expose their functionality via JMX MBeans, allowing for user interaction using a JMX console, which acts as a replacement for the ToolNet Desktop. This part of the prototype is described in Section 6.4.3.5.4.

Further, advanced concepts such as Sessions (see Section 5.3.3) or Projects (see Section 5.3.4) have been left out, because this would require migrating core ToolNet components such as the SessionManager or ProjectManager, which means a significant reengineering effort. On the other hand, the *ToolNet/JBI*-architecture is designed for extensibility and facilitates a smooth migration of the ToolNet framework in an incremental manner, component by component, as mentioned in Section 6.2.5 before.

Another aspect that is covered only rudimentary is the dynamic query functionality combined with generic *ServiceTypes* in ToolNet, as described in Section 5.3.5. The prototype supports only one CoreService, the PresentationService `OBJECT_HIGHLIGHT`, which is supported by the DOORS Adapter and the prototype UI. Being limited to one Adapter and a single Service leaves almost no room for ToolNet's query-functionality, except for the DOORS *ServiceEngine* (see Section 6.4.2.2), which uses a JBI Endpoint Query to find the DOORS BindingComponent (see Section 6.4.2.1).

6.4. Realization

The following sections describe how the goals identified above were solved with the new, JBI-based architecture and how it was successfully applied to a concrete integration scenario with the example of a prototype. While this section focuses on the parts and concepts needed for integrating COTS applications like Telelogic DOORS in a standards-based way, they can easily be reapplied for integrating tools with entirely different interfaces, while still benefiting from the new architecture and its standards-based approach. This will be covered in Chapter 7, together with a comparison to the current ToolNet architecture.

6.4.1. Analysis

Based on the requirements gathered in Section 4.1 and the goals outlined in Section 6.2, and looking at the current move from a proprietary component-architectures to OSGi (also ToolNet follows this direction), it was

a clear consequence that the new architecture should be based on open *enterprise integration* standards where possible. As elaborated in Section 3.3 and Chapter 4, the problems faced by desktop tool integration solutions like ToolNet are not uncommon in the enterprise, and while there are differences, there is much advantage in applying proven standards and solutions from enterprise integration to the special problems of desktop tool integration. This allows reusing existing integration solutions as a base for the new solution, and concentrating on those parts and concepts that are special to the desktop domain, like integration at the user interface level or Tool Adapters that wrap proprietary interfaces to COTS applications.

6.4.1.1. JBI as the Underlying Architecture

At the core of the new architecture lies the *Java Business Integration* (JBI)-standard, which is introduced in Section 4.2. JBI allows the integration of existing COTS applications to a common service backbone by exposing proprietary interfaces as common services. By moving to a higher level integration standard, the existing, proprietary ToolNet Service infrastructure can be fully transformed into a standards-based and open integration backbone. JBI builds on existing standards like WSDL for service description and lookup, and applies WSDL message exchange patterns. JBI uses XML for interoperable message exchange and makes use of enterprise integration patterns for COTS integration.

JBI also facilitates a more loosely coupled and reusable Adapter architecture: External applications like Telelogic DOORS can be accessed by implementing a custom *Binding Component*, which resembles the *ToolNetSide Adapter* in ToolNet, and one or more *Service Engines*, which relate to *ToolNet Services* (see Section 6.4.3).

The architecture can be easily implemented by using an open-source ESB with support for JBI-components as a runtime. Apache ServiceMix is the leading open source JBI ESB that was designed around the JBI specification and supports all aspects of the standard. This made it an ideal choice for the prototype implementation, as detailed below.

6.4.1.2. Apache ServiceMix ESB as the Service Backbone

Apache ServiceMix [ServiceMix] is an open source JBI implementation by the Apache Software foundation that uses JMS (through Apache's ActiveMQ message queue) for implementing the messaging backbone. It features support for all four JBI MessageExchangePatterns and already provides a wide array of JBI *BindingComponents* and *ServiceEngines* that can be reused for implementing integration solutions. It is important to note that these components can be deployed into any server-runtime that supports the JBI standard, and at the same time, components from other JBI-compliant runtimes can be used with ServiceMix. As long as components follow the JBI specification, which is verified and enforced by any compliant runtime, the developer (and user) is free to chose the JBI implementation that best meets project needs.

ServiceMix was chosen over alternative implementations because of the reasons given in Section 4.4.2, most importantly because it is designed from the ground up to embrace JBI, whereas alternatives like Mule or JBossESB just connect JBI-components as external endpoints or as additional service layers modeled upon a proprietary backbone. Also, alternative implementations like OpenESB or PEtALS offer limited tooling and community support for developing custom components.

6.4.1.3. Adapter Analysis: JNI, JCA and finally JNA

There are two standard ways in Java to integrate external, non-Java resources, covering the “last mile of integration” and reaching through to existing COTS tools: the *Java Native Interface* (*JNI*) for integrating native (C/C++) code, and the *Java Connector Architecture JCA* for integrating enterprise applications and external data resources.

In the first stage of the prototype design, both approaches were considered, but as shown below, they did not meet the requirements regarding increased reusability and adaptability to cater for changes in tools or ToolNet

Services. Also, both approaches impose additional rules and requirements that did not align well with the JBI-based service-oriented Adapter architecture. Finally, a third approach, *Java Native Access* (*JNA*), was successfully pursued, which is described at the end of this section.

6.4.1.3.1. Using the Java Native Interface (JNI)

JNI [Liang1999] is introduced in Section 3.2.2.2 and its usage in ToolNet is covered in Section 5.4. Therefore it is covered only briefly here, showing the current implementation and evaluating its relation to the prototype.

Using JNI limits flexibility of Adapters as it requires the development of a non-Java wrapper that accesses the native library or application. This creates a tight coupling to the tool interface and impedes reuse of Adapters. Using JNI in Adapters also results in a static architecture that prohibits quick adaptation to changes in requirements, tools, or the ToolNet framework itself, as new Services are introduced or tool interfaces change in the course of upgrades. As a result, other options were considered and the current implementation of the `ToolNet-Side Adapter` was not reused.

6.4.1.3.2. Using the Java Connector Architecture (JCA)

Developing a JCA Resource Adapter (see Section 3.3.7.1 for a general introduction to JCA) is not as complicated as writing a JNI wrapper and native library by hand, as only Java is involved, but the JCA specification limits the possibilities of tool Adapters by imposing strict contracts, and it is mainly targeted at enterprise information systems like databases. By having to follow certain security, threading and communication contracts, Adapter developers are not really free in choosing the integration method best suited for a particular tool. Also, installation of JCA Adapters is not fully standardized, as they can either be run in *managed mode* inside an application server or standalone, in *unmanaged mode*. This requires manual adaptation of individual Adapters to the target environment, which would result in a less dynamic, less transparent, and more tightly coupled integration architecture, which conflicts with the requirements defined in Section 6.2.

The main reason not to use JCA is however is of evolutionary nature, as explained in Section 4.2.2.2: The *Java Connector Architecture* was designed to solve specific integration problems in the enterprise Java world, targeting *Enterprise Information Systems* (EIS) like SAP or databases. The realization of external connectivity is fully covered in the JBI specification by introducing a more general concept of *BindingComponents* (see Section 4.2). By implementing a BindingComponent that connects to the tool's API, existing tools can be integrated with a maximum of flexibility, while still retaining a common, service oriented architecture, which is represented by the JBI infrastructure. This is made possible by the twofold role of BindingComponents: One part is directed at the external tool and implements whatever proprietary mechanisms are necessary to enable communication using the tool's proprietary protocol, whereas the other part is realized a common JBI component that provides a service-oriented façade for the tool's functionality, making it accessible to other JBI components in a transparent way as a set of services.

While JCA might not be the right choice for a general ToolAdapter as part of the new architecture, it is certainly possible to use JCA Resource Adapters together with the proposed solution where it makes sense, such as in situations where JCA-Adapters are already provided by tool vendors, or where existing implementations could be reused. This is possible by accessing the JCA Adapter through a corresponding JCA *BindingComponent* that exposes the ResourceAdapter as an additional binding in the JBI infrastructure. As a BindingComponent, it translates *NormalizedMessages* received from other JBI components via the `NormalizedMessageRouter` to method invocations on the JCA Adapter, and at the same time it provides call back functions or listener *Threads* for receiving messages from the JCA Adapter, which are then translated into NormalizedMessages that can be transmitted inside the JBI environment. Apache ServiceMix provides a JCA container[1] that acts as a “lightweight” *ServiceEngine* and communicates via JMS message queues. Sun provides a complete JEE ServiceEngine [Sun2006] that integrates existing EJBs into a JBI infrastructure.

[1] using Jencks, see `http://servicemix.apache.org/jca.html`

6.4.1.3.3. Using the Java Native Architecture (JNA): Final solution

JNA's design aims to provide native access in a natural way with a minimum of effort. No boilerplate or generated code is required. While some attention is paid to performance, correctness and ease of use take priority.
--from the JNA project homepage

A less known but very efficient solution for realizing interoperability between Java and native code (such as C or C++-libraries) is provided by the *Java Native Architecture* (*[JNA]*). Integrating a native library only requires the definition of a Java interface that contains the methods and structures provided by the library. For every native type, a corresponding Java type is used. More complicated mappings, such as complex types, pointers to pointers, by-reference arguments or function pointers are handled as special types provided by the JNA library. Also call-back functions are supported by defining an interface which extends JNA's `Callback` interface and contains a single method named `callback()`. JNA is already successfully applied in several projects such as [JRuby] or a gstreamer-java[2].

Internally, *JNA* uses a small JNI-stub, `jnidispatch`, to dynamically access native library functions and structures at runtime. The native library is transparently loaded into memory, and different forms of function mapping are applied, depending on the operating system and library. For the low-level work of determining the actual function names from symbols exported by the native library, the `libffi` library[3] is used. FFI stands for *Foreign Function Interface* and provides an abstraction for various calling conventions used in different operating system environments, e.g., the `Win32StdCall` calling-convention.The resulting function and structure names are then mapped to the Java interface defined earlier, and made available through a `Proxy` object that is used as a reference for the native library on the Java side. This eliminates the need to write native wrapper code or having to generate headers and native library stubs, which makes integration much more straightforward and significantly reduces development time and maintenance cost.

An overview of mappings provided by JNA is given in Table 6.1 below.

Native Type	**Java Type**
char	byte
char*	String
int	int
long	NativeLong
long long	long
void*	Pointer
size_t	IntegerType(Pointer.SIZE)
struct	Structure
function(char **buffer_p, int* len_p)	function(PointerByReference buffer_p, IntByReference len_p)

Table 6.1: Mapping native functions and types to Java with JNA

Compared to the necessary steps involved with JNI (see Section 3.2.2.2), which also requires writing a wrapper for the native library in C, JNA reduces the complexity and eliminates the need for writing non-Java code. Example 6.1 shows an example for accessing the system `C` library using JNA:

[2]see the gstreamer-java project home [http://code.google.com/p/gstreamer-java/] at GoogleCode
[3]An introduction to libffi is available at The libffi Home Page [http://sources.redhat.com/libffi/], recent versions are distributed with the GNU Compiler Collection [http://gcc.gnu.org/]. Libffi is also bundled with the JNA distribution, including some documentation.

Example 6.1: Wrapping a native library in Java using Java Native Access (JNA)

```
public interface CLibrary extends Library {❶
   CLibrary INSTANCE = (CLibrary)Native.loadLibrary("c",
                                CLibrary.class);❷
   int atol(String s);❸
}
```

❶ extend the Library interface to define a standard library with the target system's default calling convention
❷ set up an instance of the native library using JNA's Native class, and cast the result to the wrapper interface for later use
❸ define a method that maps to an equivalent native library function

It is not necessary to define a complete mapping for the native library, only the needed functionality has to be mapped. Also, custom names for methods and structures may be used, e.g., to accomodate the Java naming convention using CamelCase. This can be achieved by implementing a custom `TypeMapper` that is passed to the `Native.loadLibrary()` call.

Once the library interface is defined, library methods, structures and constants can be used just like normal Java class members, utilizing JNA's native mapping. For example, the following code in Example 6.2 can be used to access a function from the `C` library included above:

Example 6.2: Accessing native functions in Java through a Proxy interface with JNA

```
public class MyClass {
   CLibrary clib = CLibrary.INSTANCE;❶
   int num = clib.atol("42");❷
   System.out.println("The magic number is: " + num);
}
```

❶ get an instance of the native library, wrapped in a stub provided by JNA (as defined in Example 6.1 before)
❷ invoke a method declared in the interface, which results in a transparent invocation of the native function by JNA

Relevant source code excerpts for wrapping the native DOORS library to be used in the prototype can be found in Appendix A.

6.4.2. Design

The design of the ToolNet/JBI-prototype is firmly based on the JBI specification [JBI] and applies the concepts defined therein where possible, aiming at a straightforward redesign of the current ToolNet architecture. This approach enables successful adoption of enterprise integration paradigms such as service-oriented integration and architectures like the *Enterprise Service Bus* (ESB) to desktop tool integration, solving the problems of the current ToolNet implementation, while staying true to the original vision and general design. This allows for a clean, gradual migration path (as shown in Section 7.4) and solves many of the goals defined in Section 6.2 without unnecessarily breaking the existing architecture by departing from existing and working design concepts and paradigms. This section describes the redesign of the ToolNet framework into a JBI-based solution, and how the individual parts of the current ToolNet architecture are mapped to the new solution for implementing the prototype integration scenario.

As illustrated in Figure 6.1 below, the core components of ToolNet can be directly translated into JBI counterparts: The external application Telelogic DOORS communicates with the *ToolSide* Adapter-part (see Section 5.3.7), which is modeled as a *BindingComponent* (named “DOORS BC”). The *ToolNetSide* Adapter-part, on the other hand, maps the tool's functionality to a common set of ToolNet services (see Section 5.3.5) and is

realized as a *ServiceEngine* (named “DOORS SE”). Users interact with the prototype using a JMX management console, which acts as a replacement for the ToolNet Desktop that was not used for the prototype.

In the same way that ToolAdapters were realized as JBI BindingComponents and ServiceEngines, the CoreServices in ToolNet can be retrofitted into ServiceEngines that provide common services to other components connected to the JBI backbone. For example, the `RelationCreationService` could be realized as a `RelationCreationServiceEngine` that manages data relations in a database (e.g., using a *JdbcBindingComponent* to handle the external connection) and provides the existing ToolNet-Services `addAnchor()`, `removeAnchor()`, etc. (see Section 6.4.2.5 below). ServiceEngines realizing ToolNetSide Adapters would then call the corresponding services provided by the RelationCreationServiceEngine whenever they need to serve an incoming user-request (received by the ToolSideAdapter) for creating or navigating relations.

This design is strongly supported by the JBI specification and ensures loose coupling between integrated components, allowing for easier adaption to changes on the tool side (e.g., new functionality or a different interface like a .NET-DLL) and also in the ToolNet-backend (e.g., new ToolNet-Services implemented by additional ServiceEngines). This results in a clean separation of protocol-level integration, making communication with tools possible in the first place, from application-level integration, which integrates at the logical level and translates between tool-specific commands and common framework services. The end result is a combined workflow for the end user, realizing the main goal of tool integration (see Section 2.3).

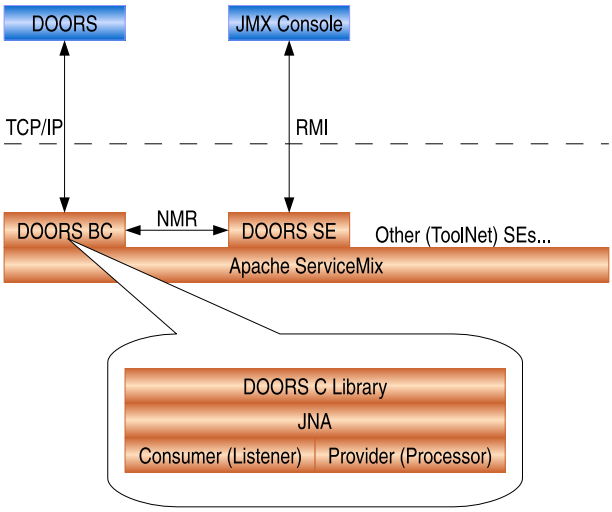


Figure 6.1: A High-level view of the prototype design showing the custom DOORS Adapter

As described in Section 6.4.1.3.3, JNA is used as a mediation layer between the external interface, provided through the DOORS C library, and the JBI-based prototype. Communication with external tools is an essential part of a desktop integration framework, and the way it is done strongly affects the quality, reliability, and developer acceptance as well as user satisfaction of the entire solution. For applications like Telelogic DOORS that provide a C-library for accessing the API (a common situation on the desktop, where web services-based or service-oriented interfaces are still rare), a thin native layer like JNA provides a simple but powerful solution.

The following sections elaborate on the design of the individual components and their role in the new architecture. The implementation is examined later in Section 6.4.3.

6.4.2.1. Using BindingComponents as Tool Adapters

BindingComponents are responsible for translating between the standardized, service-oriented communication, using XML-based NormalizedMessages inside the JBI infrastructure, and any proprietary communication necessary to interact with external tools. They realize integration on a protocol level and act as a mediator between non-JBI resources and the JBI message bus. Incoming data received from external applications is transformed

into NormalizedMessages and sent over JBI's NormalizedMessageRouter, whereas NormalizedMessages are converted into an appropriate form for communicating with the external application. The actual content of the message is never interpreted in any way other than necessary for transmission – BindingComponents only deal with messages and convert them into the target format, in the same sense that a router only works with packages and looks at their headers, leaving the actual content (here: the “message payload”) untouched (e.g., an outgoing DXL script, or an incoming ToolNet ServiceName, when viewed from the JBI runtime). Further processing on the logical level is handled by *ServiceEngines*, which are explained in the next section.

Realizing ToolNet-Adapters as JBI BindingComponents is a straightforward solution since JBI does not assume a homogeneous or Java-only infrastructure. The specification explicitly defines components that integrate external resources and protocols, but at the same time they are fully integrated into the common service bus and able to participate in message exchanges with other JBI-components in a service-oriented manner. When connecting tools to the ToolNet/JBI-infrastructure using BindingComponents, the tool's functionality is exposed in the form of Services using WSDL-mappings, as defined in the JBI specification (and also explained in Section 4.2), so that other components on the bus can transparently lookup and utilize the functionality provided by integrated tools. Because BindingComponents are normal JBI components that just have a special role of bridging from internal to external communication channels, they can also make use of other Services available on the bus as needed, e.g., for message translation, processing or routing. This level of integration is not possible, for example, with JCA ResourceAdapters, as mentioned in Section 4.2.2.2.

In the prototype, the ToolSide DOORS-Adapter is realized as a custom BindingComponent that communicates with the DOORS application using the C-library interface provided by DOORS through the JNA library, which was introduced in Section 6.4.1.3.3 (see Section 6.4.3.5.1 for implementation details). Inside the JBI container, the BindingComponent provides a service (specified again by a WSDL-definition) for sending commands to DOORS, thus hiding the necessary proprietary communication-mechanisms under a service-oriented façade. This part of the communication, from JBI to DOORS, is handled by the *Provider*-part of the BindingComponent, as shown in the callout box in Figure 6.1. Incoming commands, on the other hand, are first translated to NormalizedMessages and then propagated to the message bus for further processing by ServiceEngines described below. This part is depicted as *Consumer* (Listener).

6.4.2.2. Using ServiceEngines as ToolNet-Services

Whereas *BindingComponents* are logically situated between the JBI infrastructure and the external application (DOORS), acting as a message translator, *ServiceEngines*, on the other hand, are responsible for interpreting, routing and transforming messages on the JBI message bus (called the NormalizedMessageRouter). They rely on BindingComponents (see previous section) for handling message exchange with external resources and only communicate with internal components connected to the NormalizedMessageRouter, using service invocations. ServiceEngines therefore offer a façade to the common set of ToolNet Services (which would be implemented by other ServiceEngines in a full implementation) and implement the MessageRouter and MessageTranslator patterns as defined in [EIP].

In the prototype, ToolAdapters (precisely, the ToolNetSide Adapter-part, as explained in Section 5.3.7), are realized as ServiceEngines and are responsible for translating between ToolNet Services provided by the framework and corresponding functionality available in external tools. Consequently, the ToolNetSide DOORS-Adapter is realized as a ServiceEngine that translates DOORS commands received from the `DoorsBindingComponent` to respective ToolNet service calls and vice versa. As an example, the `HIGHLIGHT_OBJECT`-Service was implemented to demonstrate a ToolNet service invocation in DOORS (see Section 6.5 for a complete description of the prototype use case).

The `DoorsServiceEngine` also realizes the JMX management interface mentioned earlier, allowing interaction with linked DOORS *RequirementObjects* using a graphical interface. The JMX-interface is covered in more detail in Section 6.4.3.5.4.

The application of ServiceEngines is not limited to ToolAdapters: also common ToolNet Services provided by the backbone (described in Section 5.3.5) can be realized with ServiceEngines, such as the RelationService, Project- or SessionManagement-Service or the Presentation-Service. ToolNet Services provided by Adapters are exposed as WSDL endpoints, so that they can be identified as targets for the implemented services upon user request, e.g., for creating relations or highlighting an object in the integrated Tool. Adapters, on the other hand, may use ToolNet Services that support them in realizing provided Services: the RelationManager-ServiceEngine would provide services for adding or removing objects to or from a relation or for querying possible targets, and Adapters would then call these services when receiving user requests to link objects through the ToolNet operations *"Add as Source"* or "*Add as Target*" in the integrated tool. This form of communication resembles the Publish/Subscribe-pattern described in Section 4.2.2.1.

6.4.2.3. The ToolNet/JBI Backbone

The current ToolNet backbone was identified as a limiting factor of the existing architecture in Section 5.5. Today, much of the custom and partially proprietary infrastructure can be replaced by mature and standards-based implementations provided by open source projects: The custom ToolNet backbone resembles an ESB but it is not as flexible and extensible in terms of message routing, clustering, security or protocol support. As a result, currently available open source ESB solutions that implement the JBI specification fully meet the requirements for the backbone of the new solution and provide ample potential for future extension.

In Section 4.4.1, Apache ServiceMix [ServiceMix] was chosen as runtime platform for the prototype, as it provides a mature and feature rich implementation that can extend and eventually replace the current ToolNet backbone. ServiceMix is also used as the runtime environment for ChainBuilderESB [CBESB], which provides a visual development environment for JBI components and is presented in Section 6.4.4.

6.4.2.4. The JMX Interface

As mentioned in Section 4.2, the JBI specification defines JMX MBeans for component management in a standardized way. In addition, custom components may add their own MBeans for advanced management access and to expose custom functionality for management. The specification defines a `ConfigurationMBean` that components should provide for additional configuration and control.

In the prototype, JMX MBeans are provided to start and stop the DOORS application, and for controlling DOORS Objects linked to ToolNet/JBI: The DOORS BindingComponent provides a ConfigurationMBean for configuring the DXL Server port where DOORS listens for incoming connections from clients, and for adjusting the DOORS client port which is used by DOORS DXL scripts (as part of the ToolNet DoorsAdapter) to connect to the prototype. The DOORS ServiceEngine uses MBeans to represent DOORS *RequirementObjects* that have been linked from DOORS to ToolNet, exposing Object attributes as MBean attributes and providing operations to invoke ToolNet Services on the Objects; in the prototype, the `HIGHLIGHT_OBJECT`-Service is exposed as a managed operation and is accessible as an MBean operation.

For interacting with the JMX interface, any JMX-compliant management console can be used, such as JConsole[4] , which is part of the Java5 SDK, or alternative solutions like [MC4J] or web consoles like jManage[5] . Finally, Sun also open sourced the previously commercial Java Dynamic Management Kit (JDMK) with Project [OpenDMK], which includes a HTML interface and SNMP interoperability. Figure 6.2 shows the JMX management interface when accessed with JConsole:

[4]see the Java SE Monitoring and Management Guide [http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html] for an introduction to JMX and chapter 3, Using JConsole [http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html], for documentation on JConsole

[5]see jManage Open Source Application Management [http://www.jmanage.org/]

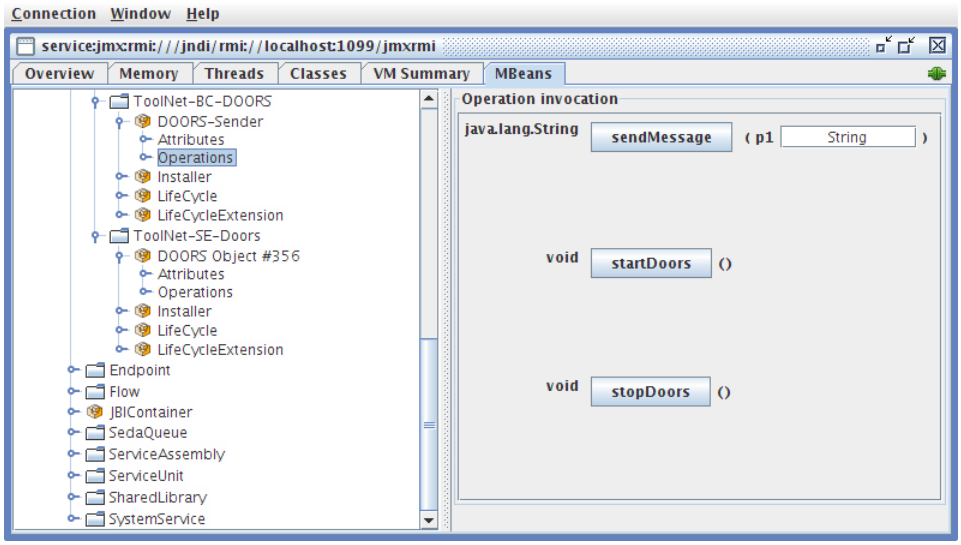


Figure 6.2: The DoorsBindingComponent MBean viewed in JConsole

As can be seen in the screenshot above, the standard Service MBeans defined in the JBI specification such as `SystemService`, `SharedLibrary` and `ServiceAssembly` are available alongside the custom MBeans defined by the `DoorsBindingComponent` and `DoorsServiceEngine`. The SystemService can be used for startup and shutdown of ServiceMix components including ServiceMix itself, the SharedLibrary is used for installing or removing shared libraries that are accessible to all JBI components on the bus. Lastly, the ServiceAssembly is used to control the lifecycle of *JBI ServiceAssemblies* (see below), such as the prototype which is packaged as a ServiceAssembly.

The `DoorsBindingComponent` MBean (named “ToolNet-BC-DOORS” in the screenshot, to comply with the naming scheme for custom components used in *ChainBuilderESB*) in the previous screenshot offers three operations:

1. `sendMessage()` sends DXL-commands to DOORS using the `DoorsBindingComponent` – this method was mainly used for testing

2. `startDoors()` offers an easy way for starting the DOORS application from within the prototype interface. Invoking this operation launches the DOORS application and automatically logs in with the default user and password, which is acceptable for a prototype but would need to be made configurable and secure for a real implementation.

3. `stopDoors()` was designed but not implemented because DOORS currently offers no way to quit the application using DXL-scripting, and killing the process is easily achieved by shutting down ServiceMix, which is the preferred way to quit the prototype. Of course, DOORS can also be quit from the application's main window by invoking File#Quit.

The `DoorsBindingComponent` MBean also exposes properties for management, available through the attributes item in the tree-view on the left. Users can change the ports used for communication with DOORS, as mentioned in the beginning of this section. These settings are only applied upon startup of the BindingComponent or DOORS, respectively (this is a limitation of the prototype, but not JMX).

The `DoorsServiceEngine` MBean in Figure 6.3 holds MBeans for Objects that have been linked from within DOORS, by invoking the ToolNet menu operation Select Object as source or Select Object as target, which defines an endpoint for a Relation between any two data model elements in ToolNet (see Section 5.3.6

for an explanation of this concept). Every linked Object is mapped to a custom MBean that is grouped under the ServiceEngine and named after the `Object ID` in DOORS. The MBean attributes describe the link type, the corresponding link description, as well as the `Object ID` and `Module ID`, which designates the *DOORS Module* containing the Object. The `AnchorType` can be either `0` (for link targets) or `1` (for link sources), as reflected by the `AnchorTypeName` attribute. The attributes `Description` and `Name` are currently not passed in by the ToolNet DXL-scripts in DOORS and are left blank in the attribute view.

Each `ObjectMBean` under the `DoorsServiceEngine` MBean implements a single operation, `highlight()`, that maps the ToolNet Service provided by the prototype to the corresponding DXL command, which is sent to the `DoorsBindingComponent` from where it is transmitted to DOORS as described in Section 6.4.2.1. More information on the implementation of the JMX interface can be found in Section 6.4.3.5.4.

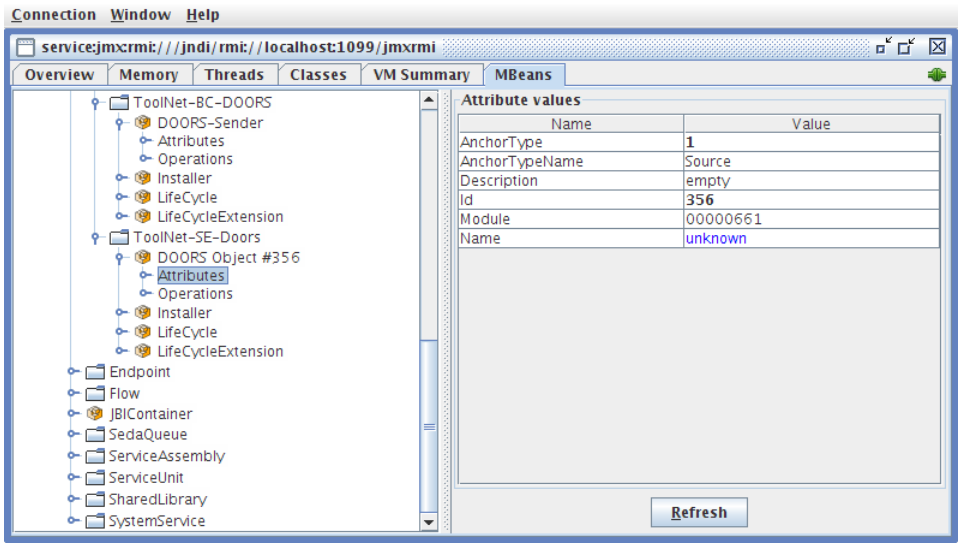


Figure 6.3: The DoorsServiceEngine MBean viewed in JConsole

6.4.2.5. Putting it all together: The ToolNet/JBI ServiceAssembly

While the previous sections focused on the individual JBI components and supporting technologies like JNA, this section examines how these components work together, viewing the prototype in its entirety, and describing the message flow from end to end.

To understand the composite solution, it is important to know that JBI BindingComponents and ServiceEngines can act as a *Consumer*, a *Provider*, or both (see Section 4.2.1). These roles are defined in the JBI configuration descriptor of the *ServiceAssembly*, which acts as a container that describes all components required for the composite application and defines the message flow between Service Endpoints (see the “connection”-elements in Example A.1 for an example). The ServiceAssembly consists of *ServiceUnits* that reference required target components installed in the JBI container and supply them with dynamic configuration at runtime, together with optional artifacts for installation into the target components (like custom DXL scripts for the `DoorsServiceEngine`). The composite application represented by the ServiceAssembly is ultimately deployed to the JBI runtime as a single package in ZIP format, including standardized JBI descriptors in XML. Figure 6.6 shows a visual representation of the prototype's ServiceAssembly. For details regarding packaging and deployment please refer to Section 6.4.4.3.

Figure 6.4 illustrates the path of a user request sent from DOORS to the prototype, showing all processing steps involved until the result is presented in the prototype's JMX interface:

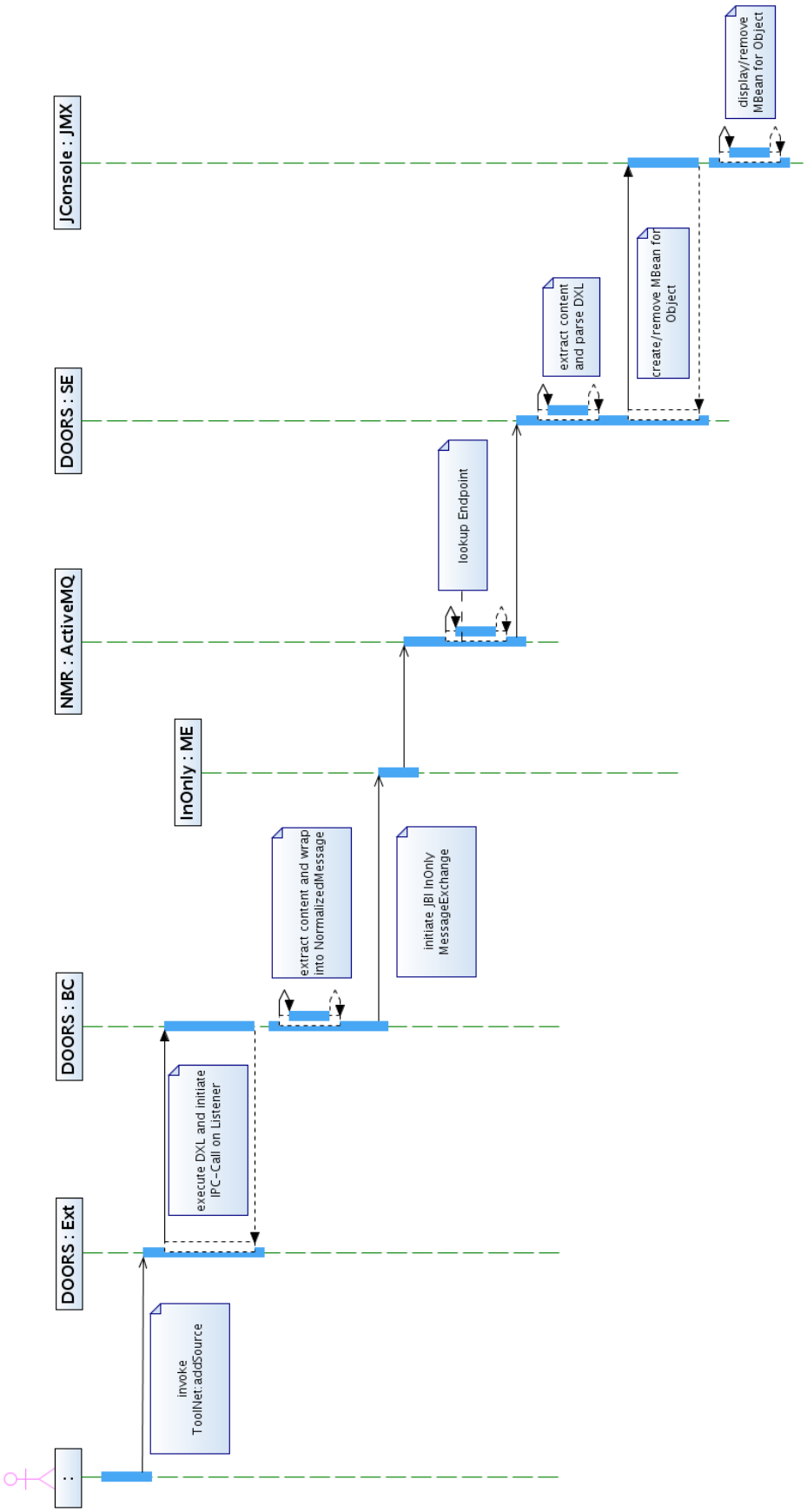


Figure 6.4: Sending a command from DOORS to the prototype

When the user invokes the ToolNet-menu items Select Object as Source or Select Object as Target, the corresponding DXL script is executed and connects to a user-defined IPC port (by default, port `5094`) by opening a `TCP/IP` connection, which is provided by the DOORS DXL API. The script sends the ToolNet-request `addAnchor()` as plain text over the channel, where it is received by the listening `DoorsBindingComponent`, which acts as a *Consumer* in this message flow, because it *consumes* Services (provided by the `DoorsServiceEngine`) from the JBI bus. There, the request is packaged into a `NormalizedMessage`, without being interpreted, and a new `InOnly` `MessageExchange` is initiated. The message is then routed to the target component as defined in the ServiceAssembly descriptor, which is the DoorsServiceEngine. The ServiceEngine, acting as a *Provider*, is responsible for extracting the original request by denormalizing the message, parsing and interpreting the request and required arguments. With this information, it creates a new `MBean` to represent the requirement object selected in DOORS, and sets the `MBean` attributes to the arguments received. The MBean's name is set to the `ObjectID` used in DOORS, sothat the new Relation can be easily indentified.

Communication in the reverse direction, sending the `highlight()` request from the prototype to DOORS, happens analogous to receiving and processing requests from DOORS: When the user invokes the operation `highlight()` on a DOORS `ObjectMBean` in the prototype's JMX interface, the request and required arguments (e.g., the `ObjectID`) are propagated to the `DoorsServiceEngine`, which now acts as a *Consumer*, because it consumes Services provided by the `DoorsBindingComponent`. Now the *ServiceEngine* is responsible for translating the user request `highlight()` into a corresponding DOORS DXL command, `showObject()`, and for initiating a new JBI MessageExchange. In the other direction, the mapping was done by the custom ToolNet menu extension in DOORS, and the MessageExchange was initiated by the BindingComponent because it received the request from DOORS. The `NormalizedMessage` containing the DXL command is then routed over the `NormalizedMessageRouter` to the DoorsBindingComponent, which now acts as a *Provider*, because it provides the necessary functionality to transmit the request to the DOORS application, using the DOORS API as described in Section 6.4.3.5.3. When the ToolNet script listening for connections inside DOORS receives the request, it parses the request's arguments and calls the corresponding DOORS API function to display and highlight the designated Object in the DOORS interface. The implementation of this usecase is described in Section 6.4.3.5.5.

6.4.3. Implementation

It’s all talk until the code runs.
--Ward Cunningham

When implementing the prototype, several goals were kept in mind to ensure a successful end result: To stay within a reasonable time frame and to avoid unnecessary reimplementation of existing artifacts, the current solution was analyzed for source fragments that could be reused in the prototype, or concepts that could be translated to the new architecture, especially regarding the communication with Telelogic DOORS using DXL scripting and socket-based inter-process communication provided by the DOORS C library. By reusing existing parts and staying with the core principles of a proven solution, a later migration would also be easier. At the same time, the new solution should meet the goals defined in Section 6.2, and demonstrate the advantages of the new architecture (as introduced in Chapter 4) in the form of an independent implementation, which is firmly based on the findings in this work (see Part I) and the technologies selected during analysis and design of the prototype. Consequently, the existing ToolNet implementation, in particular the *DoorsAdapter* (ServerSideAdapter and ClientSideAdapter), and the ToolNet DXL scripts used in DOORS were inspected for possible reuse, carefully weighting the benefits and drawbacks against the requirements set out above. The end result is a mixed approach were one part of the Adapter was reused and the other part was completely replaced.

6.4.3.1. Evaluating the current solution for reuse

The `DoorsAdapter` currently used in ToolNet follows the general design principles of the Adapter architecture outlined in Section 5.3.7 and is divided into two logical components: a *ToolSideAdapter*, which realizes the

actual integration with DOORS using socket-communication and DXL, and a *ToolNetSide* Adapter, which communicates with the ToolNet backbone. The ToolSideAdapter is realized as a set of DXL scripts (DOORS' native scripting language) that are integrated into the DOORS interface: a ToolNet menu provides ToolNet Services to the user over a familiar interface, and a ToolNet window acts as an additional palette that offers quick access to common ToolNet commands and displays information about selected requirements Objects. By registering custom ToolNet DXL scripts in DOORS, the Adapter automatically is called when DOORS receives commands from the ToolNetSide Adapter.

The ToolNetSide DoorsAdapter consists of several parts: the class `DoorsAsServerLink` opens a client connection to the DOORS DXL server, using the *Java Native Interface* (JNI, see Section 6.4.1.3.1) to access a wrapper `DLL` that calls the native DOORS C library, which provides the necessary API functions for inter-process communication with DOORS. This part of the Adapter is tightly coupled to the current ToolNet architecture and backbone (implementing several ToolNet interfaces and following certain implementation patterns), which was also identified in Section 5.5 as one of the major problems in the current implementation. The class `DoorsSocketToolAsClientLink` realizes the server connection and processes incoming requests from DOORS (sent by the `ToolSideAdapter`, that is the custom DXL client scripts).

Support for custom scripting languages in DOORS

It is worth noting that the DOORS 7.1 API provides functions for implementing a custom DXL-like language that could be used to build a scripting host on the ToolNet side similar to DOORS. Using this method, the Doors Adapter on the ToolNet side would register functions and data types needed for integration with ToolNet, and the DOORS-side Adapter (the DXL scripts) would then utilize these functions to access ToolNet Services. This creates links between DOORS and ToolNet, following a client/server model, where active links relate to the client side and passive links constitute the server side of the connection, as explained in [DOORSAPI]. While this method allows function-based integration using native data types, it has not been used in the current ToolNet-communication.

The ToolSide DoorsAdapter sends ToolNet method-invocations as Strings to the ToolNetSide DoorsAdapter, which parses them and calls the matching ToolNetService. In the other direction, the ToolNetSide DoorsAdapter implements the ToolNetServices supported (by extending the appropriate ToolNet interfaces provided by the framework), translates them to corresponding DXL scripts, and sends them to the DOORS DXL server for further processing by the ToolSide DoorsAdapter. An example of the communication between ToolNet and DOORS is given in Section 6.5.

6.4.3.2. Final Solution

Because the *ToolSide* DoorsAdapter only consists of DXL scripts that communicate over Sockets, independently of the existing ToolNet API, the existing DXL scripts and the ToolNet commands exchanged by both Adapter parts can be fully reused in the new implementation. The existing ToolSide DoorsAdapter communicates transparently with the new prototype. This eases migration, as no changes are required to work with the new implementation, and also avoids the duplicated effort of reimplementing the DXL scripts and interface elements for DOORS integration.

The *ToolNetSide* Adapter part however was entirely replaced by a new implementation, as it could not easily be integrated into the prototype without drawing in many dependencies of the current ToolNet implementation. This would have sacrificed several goals defined in Section 6.2, mainly that of a independent implementation, and also would have made it harder to show the advantages of the new approach, such as ease of Adapter development. By contrast, a clean JBI-based approach, using a dedicated `DoorsBindingComponent` (see Section 6.4.3.5.3) and `DoorsServiceEngine` (see Section 6.4.3.5.5), allowed a fresh implementation of the needed custom

components, designed from the ground up to embrace service-oriented concepts and message-based integration. The new Adapter implementation also serves as a reference implementation for further ToolAdapters, as they can be adapted to other needs with reasonable effort, which is shown in Section 7.1.

6.4.3.3. Comparing the two implementations

Migrating the current implementation of the DoorsAdapter and other parts of the ToolNet framework was identified as a potential goal but left as a separate project as it was too complex for the initial prototype implementation (see Section 7.4). However, because the new implementation follows similar patterns and concepts from a high-level perspective, migration should be straightforward, as the architecture is “compatible” to the existing ToolNet architecture: e.g., the concept of a *ToolSide* and a *ToolNetSide* (JBI) Adapter part is common to both approaches. Also, the new implementation of the ToolNetSide Adapter part can be mapped onto the old approach: the existing DoorsAdapter is logically subdivided into a Tool-specific and a ToolNet-specific part: the first part handles communication with the external tool, whereas the latter translates between ToolNet Services and DOORS commands. This separation can be found in a similar way in the new implementation, with the notable difference that now a standardized approach is applied, using JBI *BindingComponents* as the Tool-specific part, handling external communication on the protocol level, and *ServiceEngines* as the ToolNet-specific part, acting as a translator on the semantic level. A detailed example for the mapping from the old implementation to the new solution is shown in Table 6.2 below.

Functionality	**Existing implementation**	**New implementation**
sending commands to DOORS	`DoorsAsServerLink`	DoorsBindingComponent (Provider part)
receiving commands from DOORS	`DoorsSocket-ToolAsClientLink`	DoorsBindingComponent (Consumer part)
translating ToolNet Services into DOORS DXL calls	`DoorsCommandEncoder`	DoorsServiceEngine (Consumer part)
translating DOORS DXL calls into ToolNet Services	`DoorsCommandParser`	DoorsServiceEngine (Provider part)

Table 6.2: Mapping the new DoorsAdapter to the existing implementation

6.4.3.4. Software Requirements and Tool Chain

The prototype was realized in Java, using JDK 5, however no Java5-specific features or libraries were used, so the implementation should be highly portable across Java-versions from version 1.4 upwards, depending on the JBI runtime and development environment used. For Apache ServiceMix 3.x, JDK 1.5 is the minimum requirement. For the JBI runtime, Apache ServiceMix 3.2.1 was used, which implements version 1.0 of the JBI specification, the latest version available.

ChainBuilderESB 1.1 (introduced in Section 6.4.4) was used as a development environment, because it offers a visual editor for ServiceAssemblies with automatic code-generation for JBI deployment descriptors and WSDL definitions, and a wizard for creating new custom JBI components. Especially the latter was important for developing the prototype, a feature that is missing from alternatives such as NetBeans with SOA Enterprise pack (see Section 4.2.3 for a short evaluation of available JBI tooling). ChainBuilder is actually a prepackaged development environment for developing JBI-solutions: it is based on the Eclipse IDE (version 3.2) and adds plugins for a visual ServiceAssembly (“Component Flow”)-editor and JBI component-wizards for creating and configuring BindingComponents, ServiceEngines and message flows. Also, custom components are included such as the TCP/IP-BindingComponent used in an iteration of the prototype implementation (see Section 6.4.3.5.2). For build automation, Apache Ant scripts are included, which generate the necessary deployment artifacts and per-

form the actual deployment of components by copying them to a destination from where they are picked up during startup of the JBI runtime, Apache ServiceMix (see Section 6.5 for a description on running the prototype).

The DoorsBindingComponent was developed and tested in conjunction with Telelogic DOORS 7.1[6] for Linux, the included C library `api.so` was used to access the DOORS API over a TCP/IP-based IPC channel (see Section 6.4.3.5.3).

6.4.3.5. Iterations

Realization of the prototype was a complex project that incorporates several new technologies and solutions yet unproven for desktop tool integration (see also Section 7.3). Consequently, the realization of the final use case was done in several iterations, each acting as a proof-of-concept for a specific part of the solution, and as a milestone before advancing to the next part. The following sections each describe a part of the implementation separately, providing a partial view on a specific aspect of the implementation, culminating in the final iteration which realizes the full use case, building on the previous iterations.

6.4.3.5.1. Iteration 1: Proof-of-concept using JNA

The first iteration was necessary for validating the design decision to use JNA as a native bridge to DOORS, which constitutes an elemental part of the prototype. As a test, a simple Java application was implemented that sends a command passed in over the command line to DOORS using the JNA library interface, and then waits for a reply on the default port for incoming DOORS commands. The latter was realized using simple Java sockets, as there is no matching API functionality in DOORS. The sending part was realized as shown in Example 6.3 below:

Example 6.3: Sending a command to DOORS using JNA

```
// initialize DOORS API library
doorslib = (DoorsLibrary) Native.loadLibrary("api", DoorsLibrary.class); ❶
// call API to connect to DOORS
doorslib.apiInitLibrary(null, null, null); ❷
// send command over the IPC channel and close connection
doorslib.apiConnectSock(port, host); ❸
doorslib.apiSend(args[0]); ❹
doorslib.apiSend("quit_");
// shutdown DOORS API
doorslib.apiFinishLibrary(); ❺
```

The DOORS API is accessed using the appropriate functions of the DOORS C library, which is described in [DOORSAPI] and used in the current ToolNet DOORS Adapter (where JNI is used to access the native library). The steps involved are:

❶ use JNA's Native class to load the DOORS C library into memory, automatically mapping the C functions to the Java methods defined in the DoorsLibrary-interface; from now on, the C-library functions can be called by invoking the matching methods in the Java interface.

❷ initialize the DOORS C library to set up the environment by calling the Java interface method defined in DoorsLibrary

❸ set up a TCP/IP based IPC channel to DOORS (using the default DOORS DXL server port 5093 on localhost)

❹ send the command String provided by the user on the command line, e.g., `ack "Hello DOORS!"` opens a dialog box with the given text in DOORS; the connection has to be closed because the IPC channel is synchronous, causing DOORS to wait.

[6] see the DOORS product site [http://www.telelogic.com/products/doors/doors/index.cfm] for information on DOORS and for obtaining an evaluation version to use with the prototype

❺ wind down the DOORS library environment

The complete Java interface to the DOORS library is available in Example A.6. As expected from research results during design, JNA proved to be a viable solution for seamless access to native libraries for use in the new DoorsAdapter. However, the underlying operating system has to be taken into account when wrapping native libraries, as function calling conventions vary between operating systems, and different libraries have to be used for the `Linux` and the `Windows` version. A common approach with JNA is to wrap the library loading in the interface and transparently provide a JNA stub that references the library appropriate for the target platform, which is illustrated in the JNA sample programs [JNA].

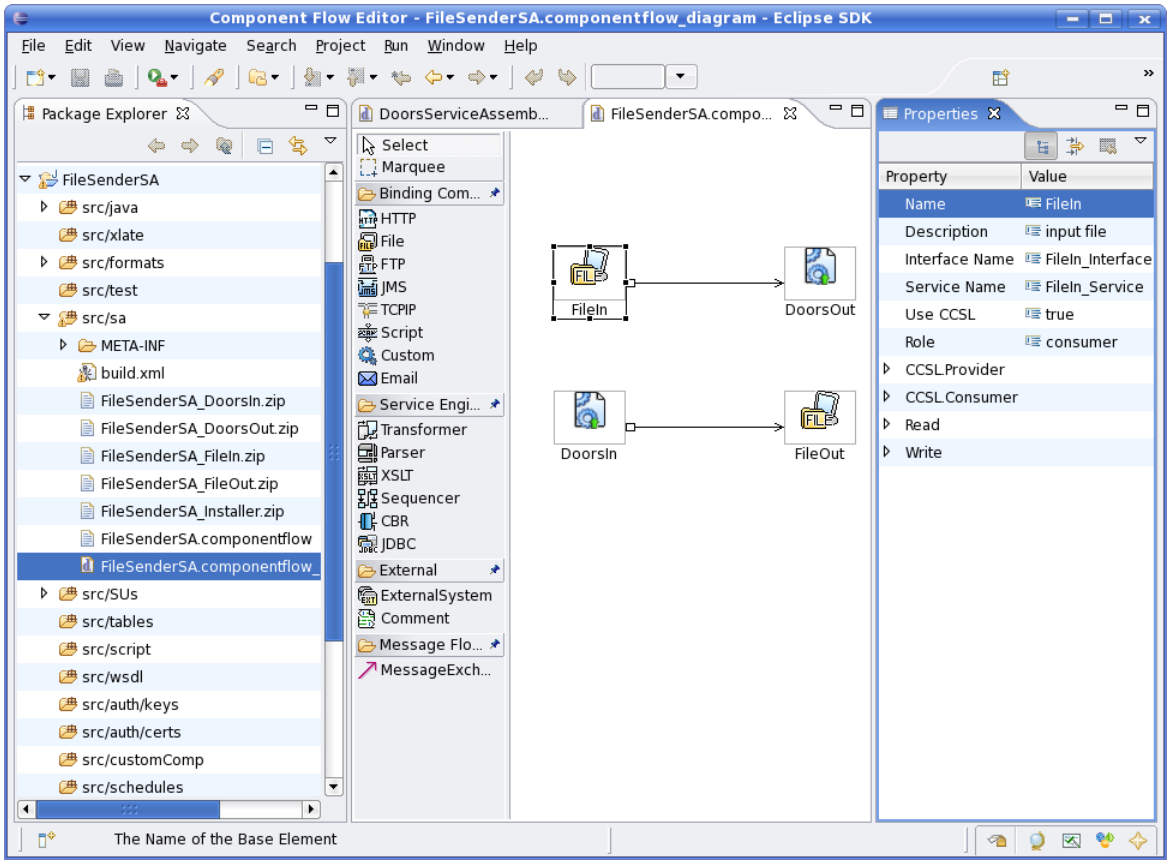
6.4.3.5.2. Iteration 2: Socket communication using a JBI TCP/IP BC

In the second iteration, a basic JBI setup was tested to verify the main concepts of the new, JBI-based architecture. It is possible to connect to DOORS using normal socket-based communication not only for receiving, but also for sending DXL commands. Using an existing `TcpIpBindingComponent` from Bostech that was committed to the open-source JBI components repository[7], a proof of concept for a JBI-based integration of DOORS was realized in a short time, as no custom component had to be developed. In this iteration, the prototype communicates directly with the DOORS DXL server using a TCP/IP-connection, without using the DOORS C library at all.

Strings can be sent and received, but they must be terminated with a carriage return, which is a requirement imposed by the BindingComponent. In the first case, the input string is read from a file, as no direct user-interaction is possible, and sent to a running DOORS instance listening on port `5093` (the default DOORS IPC port for client-connections). In the second case, input is received from DOORS on port `5094` and written to a file. The file handling is realized by a separate `FileBindingComponent` which is included by default in the ChainBuilderIDE. This component provides Services for writing out NormalizedMessages it receives over the JBI NormalizedMessageRouter to the filesystem (when configured as a *Provider*), and for polling directories for input (when configured as *Consumer*), converting the file's content to a NormalizedMessage and sending it over the JBI message bus.

The corresponding JBI ServiceAssembly is illustrated in below, where FileIn is acting as a Provider, reading files from an input-directory and transferring it to the TCP/IP-BindingComponent DoorsOut which sends the file content as a DXL script to DOORS. DoorsIn and FileOut realize the opposite direction of the communication between the prototype and DOORS, with the Provider and Consumer-roles swapped between the two components:

[7] see the project's homepage at Open JBI Components [https://open-jbi-components.dev.java.net/] and JBIWiki-Components [http://wiki.open-esb.java.net/Wiki.jsp?page=Jbicomps] for a list of downloadable components, including the TCP/IP-BC [http://wiki.open-esb.java.net/Wiki.jsp?page=TCPIPBC]



Communication with DOORS using an existing JBI TCP/IP BindingComponent
Figure 6.5: JBI ServiceAssembly for Prototype Iteration #2

6.4.3.5.3. Iteration 3: Implementing a custom DOORS BindingComponent

In this iteration, the connection to DOORS is implemented using a *custom* JBI BindingComponent that handles the communication with DOORS over the proprietary C-API interface using JNA. This was the most important and also the most complex iteration as it combines the previous two iterations, building on the successful use of JNA in the first iteration and on the JBI ServiceAssembly developed in the second iteration.

Integration with other Tools is described in the DOORS API Manual [DOORSAPI:21]. The prototype (like the original ToolNet implementation) only communicates over the library-function `apiSend()` and uses normal Java `Sockets` for receiving commands from DOORS. For this iteration, a simple test script was used that sends a message from DOORS to the prototype, to verify that incoming DOORS commands are correctly received.

For sending commands to DOORS, the corresponding DXL script is placed in a folder that is watched by a `FileBindingComponent`. This existing BindingComponent takes the contents of the File, wraps them into a JBI NormalizedMessage and sends it to the MessageRouter. The folder-location and other parameters can be configured either during designtime in the ChainBuilder “ComponentFlow”-editor, or at runtime by using a JMX management console like JConsole for adjusting the BindingComponent's managed attributes (see Section 6.4.3.5.4).

The JBI ServiceAssembly is set up sothat outgoing messages from the existing `FileBindingComponent` are routed to the new `DoorsBindingComponent`, from where they are sent over the wire to the external endpoint. The component flow diagram looks similar to , only now a custom DoorsBindingComponent is used instead of the existing TCP/IP-BindingComponent.

Outgoing communication is implemented in the *Provider* part of the custom DoorsBindingComponent as shown in Example 6.4 below (error handling and unused parameters were left out for the sake of clarity):

Example 6.4: Sending a DXL script taken from a NormalizedMessage to DOORS

```
public void processInMessage(NormalizedMessage in) throws Exception { ❶
   doorslib.apiConnectSock(5093, "127.0.0.1"); ❷
   // get message content string
   NormalizedMessageHandler nmh = new NormalizedMessageHandler(in); ❸
   Source src = nmh.getRecordAtIndex(0); ❹
   if (src instanceof StringSource) {
      StringSource strsrc = (StringSource) src;
      String dxl = strsrc.getText(); ❺
      // send in message to DOORS
      DoorsEndpoint.doorslib.apiSend(dxl); ❻
   } else {
      // got unexpected Source format, not a DXL-command
   }
}
```

When the *Provider* receives messages from the JBI MessageRouter, the method `processInMessage` ❶ is called by the JBI runtime. For this, the BindingComponent extends the `ProviderProcessor` superclass from the ChainBuilder CCSL-library (see Section 6.4.4 for details on development with the ChainBuilderIDE). The superclass handles communication details of JBI NormalizedMessage-processing, like DOM transformation and XML processing, and passes on the generated JBI `InOnly` Message for further processing. This saves component developers from some of the ground work necessary to handle JBI MessageExchanges and helps them focus on the actual application logic.

For connecting to DOORS, the BindingComponent sets up a socket connection to the DXL server port ❷ using the JNA library wrapper. The actual DXL command that should be sent to DOORS is attached to the JBI NormalizedMessage, so the method uses the NormalizedMessageHandler helper class (again provided by the ChainBuilder CCSL library) to process the input message ❸. From the message, it retrieves the message attachment ❹, and extracts the contained DXL command ❺. Finally, the command string is sent to DOORS using the API function `apiSend` ❻.

Receiving commands from DOORS is handled by the *Consumer* part of the DoorsBindingComponent in a separate `Receiver`-Thread, the `DoorsConsumerListener`, which implements a Server connection for incoming DOORS requests using normal Java ServerSockets and then reads the input using Java Sockets. When the BindingComponent is started by the runtime, the Receiver-Thread starts listening for incoming socket connections from DOORS. Everytime a new connection is established, a new JBI MessageExchange is set up and the command received is wrapped in a `NormalizedMessage` that is sent over the NormalizedMessageRouter to the `FileBindingComponent`, as configured in the ServiceAssembly. When the FileBindingComponent receives new input, it writes the String contained in the NormalizedMessage to a file, which can then be viewed by the user. The command itself is not interpreted in this iteration, only message transport and translation is realized.

On the DOORS side, a DXL script has to be executed to open up a server connection for incoming requests sent by the prototype. This is done using a startup DXL script which has to be placed in the `$DOORSHOME/lib/dxl`. To send a command to the prototype, the user has to invoke the menu command Tools#Edit DXL and then load or type in a valid DXL-script such as the following simple echo-command in Example 6.5 below, which sends a simple text to the prototype listening on port `5094`.

Example 6.5: Opening a simple dialog in DOORS from Java using JNA

```
IPC javaSocket;
javaSocket = client(5094, "127.0.0.1");
if(! null javaSocket) {
```

```
    send(javaSocket, "Hello JBI!\n\r");
    delete(javaSocket);
} else {
    infoBox("no network connection");
}
```

The source code of the `DoorsBindingComponent` (including the Listener that has been omitted here) is shown in Section A.2.1. The article [JBIDev] in the OpenESB-Wiki gives a detailed description of the steps necessary to create a custom BindingComponent, including an example with full source code.

6.4.3.5.4. Iteration 4: Implementing the JMX interface

This iteration implements a user interface to make the prototype more realistic and to allow for direct user interaction. The interface used in the current ToolNet implementation, the RCP-based ToolNet desktop (see Section 5.3.2), was too complex and would have created too much dependencies on the current implementation. Instead, JMX was chosen as a lightweight and straightforward solution that allowed to add user interaction without having to write user interface code just for the prototype. The solution is described in Section 6.4.2.4 and has been successfully validated in this iteration.

As mentioned in Section 4.2.1, and defined in the JBI specification, chapter 6 “Management”, components may register optional `ExtensionMBeans` to provide additional possibilities for management and configuration at runtime. In JBI, JMX is mainly used for administrative tasks like component lifecycle management or installation of shared libraries, which is handled by explicitly defined MBeans like the `InstallationServiceMBean`, `DeploymentServiceMBean` or `ComponentLifeCycleMBean`. This allows for runtime configuration of BindingComponents, ServiceEngines and other infrastructure components available in the JBI runtime implementation, e.g., Apache ServiceMix[8].

The prototype relies on standard installation and deployment services implemented by the runtime as required by the specification, and implements additional MBeans for configuration and control of the DoorsBindingComponent. As the JBI specification does not define standard conventions for custom component MBeans regarding naming and how the configuration and advanced capabilities should be exposed to management, the current practice promoted by Sun is to use a `ConfigurationMBean`[9] that provides advanced functionality not accessible over the standard MBeans.

In this iteration, the `DoorsBindingComponent` from the previous iteration has been extended to provide a custom MBean that is registered as an `ExtensionMBean` as required by the specification. The new *DoorsBindingComponent MBean* (called “ToolNet-BC-DOORS”) exposes the configuration necessary to interact with DOORS and allows the user to send DXL commands over the IPC channel to the external application.

The custom MBean is registered with the JBI MBeanServer during component initialization, in the `ComponentLifeCycle.init()` method, and from then on it is immediately accessible from JMX management consoles like JConsole. In the `ComponentLifeCycle.shutdown()` method, the MBean is unregistered again, so after the BindingComponent is stopped, configuration and control is no longer possible.

The DoorsBindingComponent MBean provides an operation `startDoors()` for starting DOORS from the JMX console, using `System.execute()`, which launches the DOORS application and passes in several DOORS command line switches to allow auto-login with a supplied user and password, as described in [DOORS:379]. It also implements an operation `sendMessage()` that takes a String argument and sends it in raw form as a DXL command to the DOORS DXL server, as implemented in the previous interation.

[8] see the page JMX Console [http://cwiki.apache.org/confluence/display/SM/JMX+Console] in the Apache ServiceMix Wiki for more information on JMX access

[9] see the article The HTTP/SOAP JBI Binding Component [http://blogs.sun.com/gopalan/entry/the_http_soap_binding_component] in Gopalan Suresh Raj's blog “Web Cornucopia” for more background information on managing custom components with the example of the HTTP/SOAP-BindingComponent, which is also explained in the article.

This iteration realizes only the *outgoing* communication in the BindingComponent, as it served mainly as a testbed for the JMX user interface. The complete use case with in- and outgoing communication and control was realized in the final iteration described below.

6.4.3.5.5. Final Iteration: Implementing the use case

The iterations described earlier lay the foundation for a service-oriented, JBI-based communication with the external DOORS application. This iteration builds on the prototype components developed in previous iterations and realizes the use case described in Section 6.4.2.5, for which it relies on the JMX interface developed in the previous iteration.

In the final implementation, *protocol-level* integration for connecting the external application DOORS is cleanly separated from the *application-level* integration of the service-oriented ToolNet interface and functional DOORS interface using DXL calls. Protocol-level integration is realized by the `DoorsBindingComponent` as described and implemented in Section 6.4.3.5.3, translating between the service-oriented communication using NormalizedMessages inside the JBI container and socket-based IPC for connecting to external Tools. For application-level integration, a new `DoorsServiceEngine` has been implemented, which replaces the simple FileBindingComponent from earlier iterations with a custom JBI ServiceEngine that translates ToolNet Service requests sent by other ToolNet-components[10] into corresponding DXL calls. The new component also allows direct user interaction, as required by the use case, by providing methods for JMX-based management-access via JMX-enabled management-consoles such as JConsole, thereby using existing management consoles as a replacement for the ToolNet desktop within the scope of the prototype use case (see the previous iteration for information on the JMX implementation in the BindingComponent, and Section 6.4.2.4 for a detailed overview of the design-aspects). Together with the custom `DoorsBindingComponent` and associated `Extension-MBean`, communication in both directions is possible at the presentation level, which resembles the original ToolNet vision using the concepts developed in this thesis.

For sending commands from DOORS to the prototype, the original ToolNet DXL scripts have been reused, only this time they transparently send ToolNet commands to the new implementation. For the prototype use case, only a few of the ToolNet scripts are needed: the common ToolNet configuration script that declares global variables for IPC and ToolNet-Adapter options, a script realizing the ToolNet window, the IPC implementation itself, and the script implementing the *RelationCreationService* (sending ToolNet-commands for creating links), and finally the *PresentationService* providing the `highlightCurrentObject()` method, which changes the module view in DOORS to focus on the specified Object, as needed for the use case. In the existing ToolNet implementation, this function is also available in the reverse direction and realized in the RelationCreationClient's `highlightObject()` method, which sends a `highlight()` request from DOORS to ToolNet. The corresponding Service has not been implemented in the prototype, as it would require manipulating the JConsole interface for implementing the necessary functionality to highlight a specific MBean, which would have bound the prototype to JConsole. Details on the scripts used in DOORS and their function are given in Section A.3.

For implementing the use case, a JBI ServiceAssembly (illustrated in Figure 6.6) has been developed that includes two instances of the custom DoorsBindingComponent developed in iteration 3 and two instances of the new DoorServiceEngine described below[11]. One instance always acts as a *Consumer* that processes an incoming request and then invokes a Service provided by the other component to complete its task. The other instance acts as a *Provider* and implements a higher level Service that is called by the *Consumer* instance and published

[10] In the prototype, the request comes from the ServiceEngine's associated JMX MBean that represents the target Object in DOORS, which directly calls the highlight()-method in the ServiceEngine. A message-based, service-oriented communication is only implemented for the ServiceEngine-Provider that receives DXL-messages from the DoorsBindingComponent.

[11] To be more precise, the ServiceAssembly contains JBI ServiceUnits that hold the required custom configuration described for later deployment to the target components. The component flow editor provides a high level design view which sees JBI components as building blocks, without the packaging details required for deployment. When building the ServiceAssembly, the editor plugin translates the design into corresponding artifacts that can be deployed to the target environment at runtime when the ServieAssembly is started in ServiceMix (see Section 6.5).

inside the JBI infrastructure. In the first case, the `DoorsBindingComponent` Consumer handles incoming requests from DOORS and then *consumes* a Service of the `DoorsServiceEngine` for translating the received command into a ToolNet Service. In the other case, the `DoorsServiceEngine` Consumer uses the `send()` Service provided by the `DoorsBindingComponent` to route the DXL command to the external application.

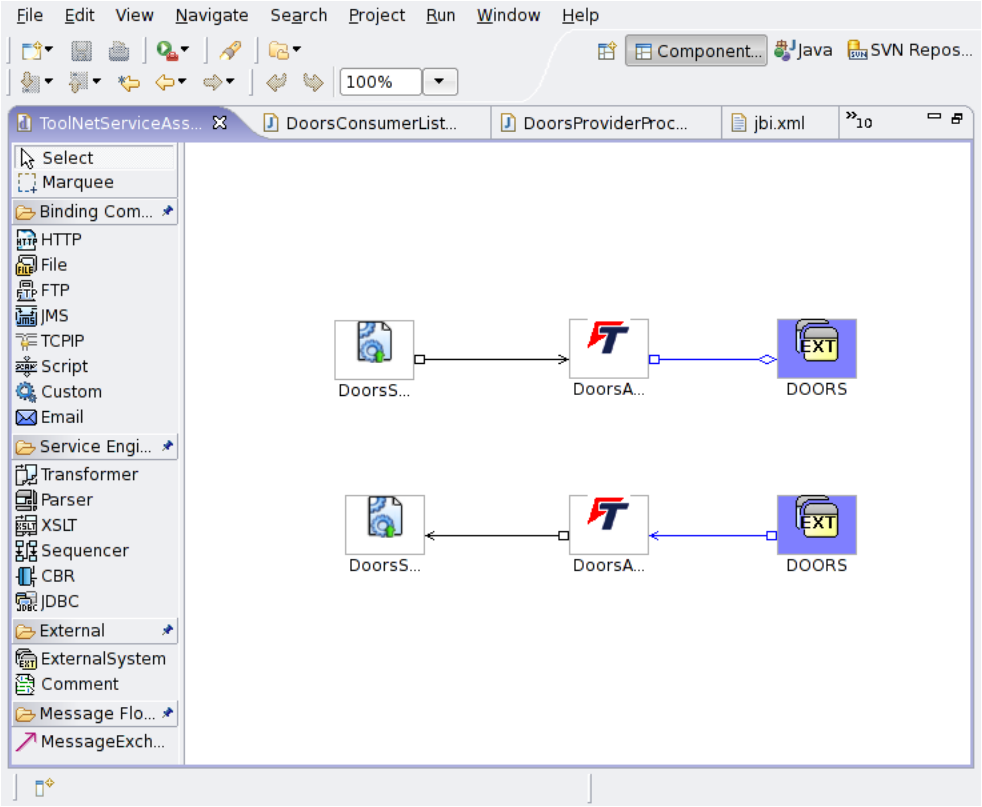


Figure 6.6: The final DoorsServiceAssembly viewed in Chainbuilder's component flow editor

As outlined in the use case description in Section 6.4.2.5, Objects in DOORS can be defined as *source* or *target* of Links by using the ToolNet-menu commands Set object as source and Set object as target. The menu commands are implemented as DXL scripts (see Section A.3) that comprise the tool-side Adapter part and are configured in the menu definition file shown in Example A.15. Because only one application is integrated with the prototype, the “source” and “target” links have no special meaning, but the link nature is reflected in an MBean attribute “anchorType” later.

When the user invokes one of the menu operations to create a Relation, the corresponding DXL script is executed and generates a request-String that is then sent over the IPC channel to the DoorsBindingComponent's Receiver-Thread, the `DoorsConsumerListener`. For a Source-Link, this request looks similar to Example 6.6 below:

Example 6.6: ToolNet-command as received by the DoorsBindingComponent

```
org.toolnet.core.model.services.IRelationCreation:addAnchor((id)["00000661","356","__NULL__","__NULL__"],
(AddAsType)"1")
```

The request is not interpreted by the `DoorsBindingComponent` but translated into a `NormalizedMessage` for transmission over the JBI NormalizedMessageRouter. The BindingComponent specifies the target Endpoint `ToolNetServiceAssembly_DoorsAdapter_In_Consumer`, which is the `DoorsServiceEngine` (introduced in Section 6.4.2.2) acting as a *Consumer*, and the target Service

`ToolNetServiceAssembly_DoorsAdapter_In_Service` (the ServiceEngine only provides a single Service that interprets ToolNet commands) so that the request is propagated to the other part of the ToolNetSide DoorsAdapter, from the *BindingComponent* that received the request to the *ServiceEngine* that interprets the request.

When the `DoorsServiceEngine` receives an incoming JBI message in the DoorsServiceEngineProviderProcessor's `processInMessage()` method, it extracts the ToolNet request-String and interprets the request and its arguments by using simple String parsing[12]. In the prototype, only the `addAnchor()` request is implemented, as required by the use case. The arguments specify the `module ID` and the `ObjectID`, which are needed to reference the linked Objects in DOORS later. The `__NULL__` arguments contain an optional name and description but are unused in this scenario. After successfully parsing the ToolNet request, the `DoorsServiceEngine` creates an `ExtensionMBean` and stores the `Object ID` and link type in MBean attributes. The new `DoorsObjectMBean` is registered under a descriptive name that includes the `ObjectID`, e.g., “DOORS Object #356”, allowing easy identification of the linked Object in the JMX console by the user. Internally, the MBean keeps a reference to the ServiceEngine's Endpoint sothat it can initiate a *MessageExchange* when it receives user input, using the Endpoint's `highlightObject()`-method.

In addition to the managed attributes that identify the linked Object and that are displayed in JConsole's attributes-view (see Figure 6.3), the MBean also provides a managed operation, `highlight()`, which implements the ToolNet *PresentationService* method `SHOWOBJECT`. When the user invokes this operation using a JMX management console, the MBean's `highlight()`-method calls the corresponding Endpoint method `highlightObject()` that implements the necessary request and JBI message handling. First, a DXL script is generated that calls the appropriate function provided by the ToolSide DoorsAdapter, shown in Example 6.7 below:

Example 6.7: The DoorsServiceEngine sends a request for highlighting an Object in DOORS

```
public void highlightObject(String module, int no) {
 String dxl="#include <addins/ToolNet/ToolNet_PresentationService.inc>;" +
   "ToolNet_IPresentation_showObject(\"" + module + "\",\"" + no + "\"," +
   "\"null\",\"null\",\"HIGHLIGHT_OBJECT\")";
 sendMessage(dxl);
}
```

In the `sendMessage()` method, the DXL script is embedded into a JBI `NormalizedMessage` which is then sent to the DoorsBindingComponent by specifying the corresponding Endpoint and Service name as message target, analogous to the processing necessary when propagating ToolNet commands to the JBI message bus in the DoorsBindingComponent, as described above. An important difference is that the Component's roles are now reversed, as the DoorsServiceEngine now acts as a *Consumer* because it invokes a Service provided by the DoorsBindingComponent (which is now a *Provider*), for sending the request to DOORS.

When receiving the message that contains the ToolNet-request, the DoorsBindingComponent does not interpret the request in any way, but only extracts the request containing the DXL script, as necessary for transmission over the IPC channel, and sends the `highlight`-command to DOORS, as described in Section 6.4.3.5.3. When the message is received by the DOORS DXL server, it calls the corresponding DXL-script `ToolNetPresentationService` provided by the ToolNet-Adapter (see Example A.20), which brings the selected Object into focus by using the DOORS API function `setSelection()`.

For a walkthrough from the end user perspective, see Section 6.5. A comparison to the current ToolNet implementation is given in Section 7.2, which also evaluates strengths and open issues of the new solution.

[12]In the current ToolNet implementation, the ANTLR [http://www.antlr.org/] grammar parsing library is used which provides advanced parsing features that were not needed for the prototype implementation, therefore it was left out to keep the prototype simple and focused on the use case at hand.

6.4.4. JBI Development with ChainBuilder ESB

"The development time and maintenance cost to manage diverse applications are reduced when business integration components are built on standards like Java Business Integration (JBI), but don't confuse the ease of using the standardized run-time components with the creation of those run-times."
--Kristen Puckett

For developing the prototype, [ChainBuilder], [CBESB] (introduced In Section 4.2.3) was selected for the following reasons:

1. *common development platform:*

   ChainBuilder is based on the Eclipse platform that is also used as the base for the current ToolNet implementation: the Java IDE is used for ToolNet development, and the RCP platform is used for the ToolNet desktop, so a possible migration is eased and existing plugins or other artifacts can be reused.

2. *rich design-time support:*

   The IDE offers a visual editor for creating composite applications out of existing *and custom* components, thereby greatly simplifying the error-prone configuration of JBI *ServiceAssemblies* by providing palettes and wizards for component configuration, and automatically generating the necessary deployment descriptors during build time.

3. *code generation*:

   ChainBuilder provides wizards for creating custom components and automatically generates associated source templates and build scripts, which reliefs developers from having to write boilerplate code and XML configuration files, allowing them to concentrate on implementing the application and business logic necessary to solve the integration problem at hand.

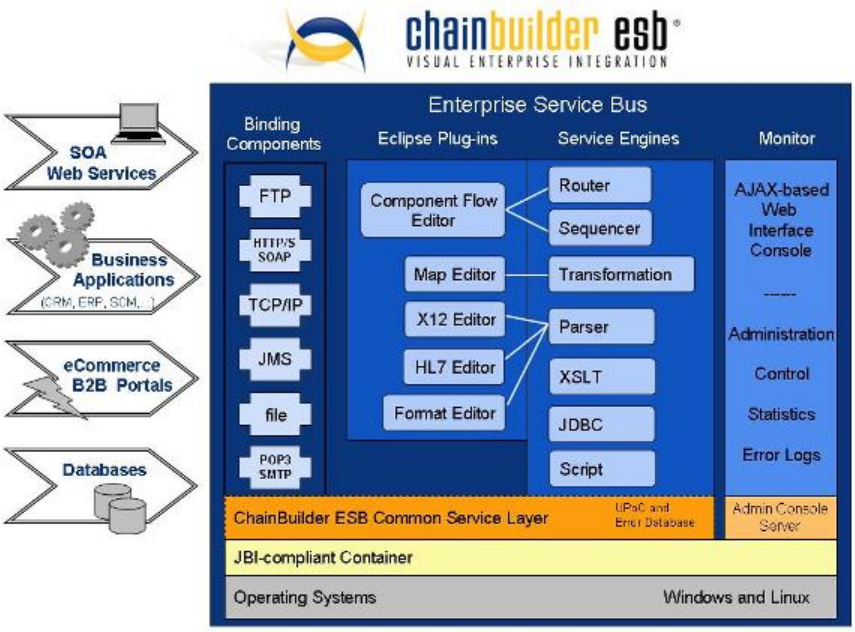
4. *support for implementing custom components:*

   Development of custom JBI BindingComponents and ServiceEngines is simplified by the included CCSL shared library as explained in Section 6.4.4.1.

5. *open source solution:*

   ChainBuilder is a pure open source solution based on open source components such as Eclipse and Apache ServiceMix. Also, the modifications and additional components (e.g., the ChainBuilder Eclipse plugins for the visual editor, or the CCSL library) are freely available under the GPL license[13]. This aligns closely with the ToolNet vision of using open standards and solutions, resulting in a stable and durable development environment.

ChainBuilder ESB provides an integrated solution for developing composite applications based on the JBI specification. The included IDE is based on Eclipse (version 3.2 as of ChainBuilder version 1.1) with additional plugins for providing advanced design-time support like a visual editor for designing JBI ServiceAssemblies (see Section 6.4.2.5) or wizards for custom component creation and configuration. Apache *Ant* is used for building the final JBI ServiceAssembly that is then deployed to Apache ServiceMix, the JBI runtime described in Section 4.4.1. The individual components that make up the ChainBuilder ESB solution are illustrated in Figure 6.7 and described below.



(from [ChainBuilder])
Figure 6.7: Schematic overview of ChainBuilder ESB

As ChainBuilder ESB is entirely Java-based, it should work on any platform with Java5 (required by ApacheServiceMix 3), but Windows and Linux are the officially supported platforms. The JBI-compliant container is provided by Apache ServiceMix 3.1 (during development, ServiceMix was upgraded up to version 3.2.1 without problems). ChainBuilder ESB *Common Service Layer* (CCSL) is implemented by the `CCSL` shared library that is deployed to the JBI runtime and explained in Section 6.4.4.1 below. For many integration scenarios, ChainBuilder provides suitable BindingComponents (such as the *TCP/IP-BindingComponent* used in Section 6.4.3.5.2 or the *FileBindingComponent* that was used in iterations 2-4), and ServiceEngines for message transformation and advanced (rule-based) routing. The prototype was however developed using custom components that were necessary for accessing the proprietary DOORS API and for translating between ToolNet requests and corresponding DXL scripts (see Section 6.4.3.5.5). For monitoring, ChainBuilder provides a web based management console that offers a streamlined interface for administration and control of the JBI runtime and deployed components, with additional statistics and alerting, similar to but more advanced than the simple web interface provided by ServiceMix[14]. The prototype does not use any web interface and relies solely on the JMX management interface provided by Apache ServiceMix, which makes the prototype independent of the runtime used, as JMX management access is required by the JBI specification (see Section 6.4.2.4 for the design of the JMX management layer and Section 6.4.3.5.4 for additional details on the implementation).

6.4.4.1. The ChainBuilder Common Services Layer

Development of custom components is described in the *Custom Components Reference* [CBESBCC], which shows the steps necessary to implement a simple ServiceEngine. Additional documentation is rare and has been gathered from ChainBuilder and ServiceMix community forums or from existing sources such as the ChainBuilder HttpBindingComponent, the TcpIpBindingComponent, and the detailed description given in Sun's OpenESB-Wiki[15].

[14] see the article ServiceMix-Web [http://servicemix.apache.org/servicemix-web.html] in the ServiceMix Wiki
[15] see the article Developing JBI Components [https://open-esb.dev.java.net/public/jbi-comp-examples/Developing_JBI_Components.html]

Because developing custom components is not covered in detail in the JBI specification[16], and requires extensive knowledge of the runtime implications of the specification, ChainBuilder ESB offers a utility library, CCSL, that aids in component development by providing extensions to the JBI API:

> ChainBuilder Common Services Layer (CCSL) is a software module in ChainBuilder ESB. CCSL provides a general service layer between JBI components and a JBI container. General services are things like centralized error handling and user-defined scripting. CCSL is designed to be inserted transparently between a JBI compliant component and contianer. This makes CCSL services available to components from other vendors.

CCSL provides helper classes and custom interfaces that makes developing JBI components more straightforward and avoids having to reimplement commonly needed functionality for every custom component. For example, the prototype implements the `ProviderProcessor`-interface and overrides CCSL methods that are transparently triggered on certain JBI events and preprocessed by the library, e.g., `processInMessage()` transparently parses the *NormalizedMessage* and returns the contained information as arguments. Also, parsing input messages and extracting attachments is simplified, avoiding the need to traverse the NormalizedMessage's DOM, as well as interacting with JBI's NormalizedMessageRouter (providing simplified `send()` methods). The CCSL library is similar to but more extensive than the Component Helper Classes provided by Apache ServiceMix[18], whereas the latter offers the possibility to implement lightweight components[19] that are easier to write than full-featured JBI components, but only available in ServiceMix and not standardized by the JBI specification.

6.4.4.2. Implementing the Prototype using ChainBuilder ESB IDE

The prototype was developed as a set of Eclipse projects, as described in [CBESBCC]: for the `DoorsBinding-Component` and the `DoorsServiceEngine`, custom component projects were created with ChainBuilder's *custom component wizard*. The generated templates and configuration files were adapted as needed, e.g. the `jbi.xml` descriptor had to be extended for including the additional shared library `jna.jar` required for native access to the DOORS C library (see Section 6.4.1.3.3). Finally, the generated class skeletons were extended with custom functionality required for the use case. This included overriding the abstract message processing methods for routing DOORS input from the `DoorsBindingComponent` to the `DoorsServiceEngine` and back. Also, additional classes had to be introduced for implementing a custom Listener Thread that receives requests from DOORS using Sockets (see Section 6.4.3.5.3). Custom `MBeans` were added for managing the DoorsBindingComponent and for interacting with the DoorsServiceEngine to access the external tool DOORS. For these additional tasks that were necessary to integrate DOORS, ChainBuilder provided useful support through the CCSL library, but implementing custom JBI message handling and integrating external libraries is still complex and error-prone. As no runtime support is provided for debugging individual components or the complete ServiceAssembly, errors in the implementation are hard to identify and require extensive logging and testing.

After the custom components were finished, a ServiceAssembly-project was created that wraps the custom component projects realizing the DoorsBindingComponent and DoorsSerivceEngine into a composite application that can be deployed into the JBI runtime, Apache ServiceMix. For this, ChainBuilder provides a wizard that sets up the ServiceAssembly-project and associated runtime configuration, including the JBI deployment descriptor `jbi.xml`. The project structure is illustrated in Figure 6.8 and shows the composite application that includes the individual components developed earlier and the necessary configuration files for configuration and deployment.

Finally, the wizard creates an empty component flow diagram that represents the ServiceAssembly, including component configuration and associated ServiceUnits for deployment (see `src/sa/ToolNetServiceAssembly.componentflow_diagram` in the figure). The diagram is edited using a

[16] this was identified as one of the weak spots of JBI 1.0 and is to be rectified in version 2.0 of the specification, see [JBI2]
[18] see the Component Helper Classes [http://servicemix.apache.org/component-helper-classes.html] page in the Apache ServiceMix Wiki
[19] see the page Lightweight components [http://servicemix.apache.org/lightweight-components.html] in the Apache ServiceMix Wiki

visual editor that allows adding the custom components developed earlier by selecting them from the component palette and clicking on the diagram. Using a component wizard, additional properties provided by custom components can be configured, as specified in the component's GUI-template, which is defined by the ChainBuilder environment. These settings can be modified later in property palettes common in the Eclipse IDE. Lastly, message flow between components is defined by drawing connections in the associated direction, according to the component's role which is set in the component wizard (or in the property palette) as either Consumer or Provider. To summarize, the ServiceAssembly-editor is an elemental and powerful part of the ChainBuilder IDE, as it provides a visual representation of the underlying JBI modules, and rich manipulation possibilities that enable dynamic configuration of needed components according to the use case at hand.

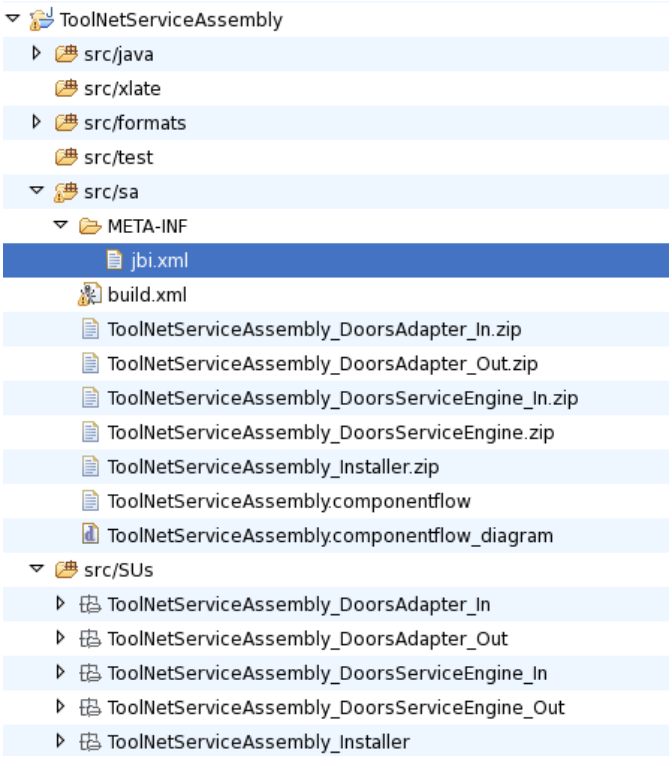


Figure 6.8: Project structure of the Prototype ServiceAssembly

For a complete walkthrough of the associated design-time development activities please refer to Section B.2, which lists the steps involved in creating and configuring the JBI ServiceAssembly for the prototype using the visual ServiceAssembly-editor of the ChainBuilderESB IDE.

6.4.4.3. Deployment in ServiceMix

Before the prototype can be run in the JBI environment provided by Apache ServiceMix, the required custom components `DoorsBindingComponent` and `DoorsServiceEngine` have to be installed into the JBI runtime. For this, they have to be packaged as defined in the JBI specification (section 6.3 “Packaging”, see also Section 4.2.1). The *component creation wizard* in the ChainBuilder IDE generates the required JBI component descriptor, `jbi.xml`, that identifies the component together with a short description, and defines provided services and implementation dependencies, like required shared libraries, the main class that bootstraps the component, or the classpath itself. For the `DoorsBindingComponent`, the JNA library needed for integrating the native DOORS C library had to be manually added to the configuration.

The wizard also generates Ant scripts that perform the actual build of the component as a JAR, and packages the result, including the aforementioned `jbi.xml` descriptor as a ZIP file that can then be copied into the ServiceMix

install-folder, which installs the component into the JBI environment. The build is initiated by selecting the component's Ant-script `build.xml` and invoking the standard Eclipse operation for running external Tools like Ant through the Run As#Ant Build menu operation.

When the components have been successfully built, they can be added to a JBI *ServiceAssembly* that defines the relationships and responsibilities of each component (as described in the previous section), acting as a container that represents a composite application, which can then be deployed as a whole in the JBI runtime, Apache ServiceMix. In the component flow editor, a shortcut menu is available that provides a Build operation for the ServiceAssembly. When invoked, the components of the ServiceAssembly are translated into corresponding *ServiceUnits* that contain the component properties and roles configured earlier. ServiceUnits are packaged as individual ZIP files, together with the JBI deployment descriptor for each ServiceUnit. Invoking Deploy in the same menu finally creates a single ZIP file containing the ServiceUnit-archives created before, and adds a deployment-descriptor (`jbi.xml`) for the ServiceAssembly. The created packages are then copied to target directories in the ChainBuilder installation from where they are picked up later when ServiceMix is started using the ChainBuilder `cbesb_run` script. This has to be done from the console, as there is no debugging or runtime support from within the IDE (see Section 6.5), which is one of the drawbacks of ChainBuilderIDE and JBI development in general (see Section 7.3.1).

Figure 6.9 shows a deployment diagram of the prototype runtime and associated execution environments, such as the JMX management console and the external application DOORS integrated by the prototype:

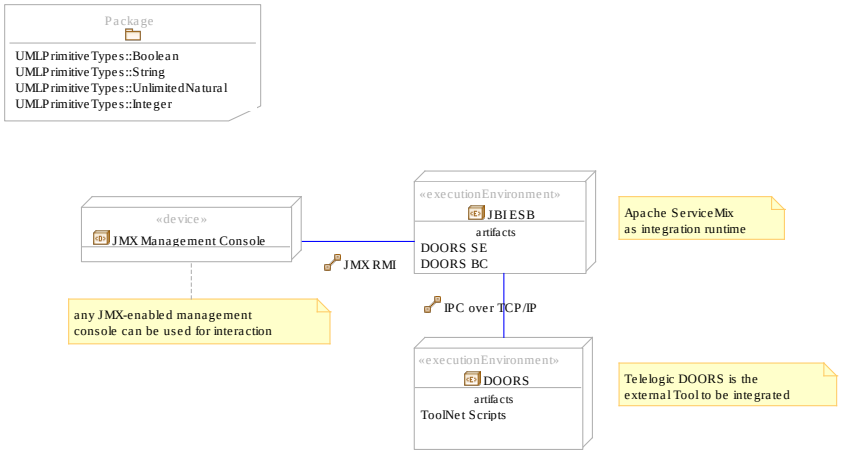


Figure 6.9: Runtime deployment overview

6.5. Running the Prototype

Apache ServiceMix can be either started standalone or embedded inside a JEE application server such as JBoss or Apache Geronimo[20], which results in a flexible approach to integration by being able to take advantage of existing resources when needed but also offering a lightweight JBI runtime where no JEE or other application server features are required, and any overhead is to be avoided, like here.

The prototype is started from the console from within the ChainBuilder install directory. First, the environment properties have to be set up by issuing the command

```
. set_cbesb.sh
```

[20]see the pages Running Apache ServiceMix [http://cwiki.apache.org/confluence/display/SM/Running] and War Deployment [http://cwiki.apache.org/confluence/display/SM/WAR+Deployment] in the ServiceMix Wiki

Then, the prototype ServiceAssembly is started by executing the following command:

```
cbesb_run ToolNetServiceAssembly
```

This starts up Apache ServiceMix and sets up the prototype's *ToolNetServiceAssembly*, as shown in Example 6.8 (log output is shortened to relevant information from the ToolNetServiceAssembly):

Example 6.8: Apache ServiceMix starting up

```
Starting Apache ServiceMix ESB: 3.2.1

Loading Apache ServiceMix from file: servicemix.xml ❶
INFO  - JBIContainer              - ServiceMix 3.2.1 JBI Container (ServiceMix) is starting ❷
INFO  - ConnectorServerFactoryBean - JMX connector available at: service:jmx:rmi:///jndi/rmi://localhost:10
99/jmxrmi ❸
INFO  - DeploymentService         - Restoring service assemblies ❹
INFO  - JBIContainer              - ServiceMix JBI Container (ServiceMix) started
INFO  - AutoDeploymentService     - Directory: install: Archive changed: processing ToolNet-BC-DOORS-1.0.jar ... ❺
INFO  - ComponentMBeanImpl        - Starting component: ToolNet-BC-DOORS
INFO  - ComponentMBeanImpl        - Initializing component: ToolNet-BC-DOORS
INFO  - AutoDeploymentService     - Directory: install: Finished installation of archive:  ToolNet-BC-DOORS-1.0.jar
INFO  - AutoDeploymentService     - Directory: install: Archive changed: processing ToolNet-SE-Doors-1.0.jar ... ❺
INFO  - ComponentMBeanImpl        - Starting component: ToolNet-SE-Doors
INFO  - ComponentMBeanImpl        - Initializing component: ToolNet-SE-Doors
INFO  - AutoDeploymentService     - Directory: install: Finished installation of archive:  ToolNet-SE-Doors-1.0.jar
INFO  - AutoDeploymentService     - Directory: deploy: Archive changed: processing ToolNetServiceAssembly.zip ...
INFO  - ServiceAssemblyLifeCycle  - Starting service assembly: ToolNetServiceAssembly ❻
INFO  - ServiceUnitLifeCycle      - Initializing service unit: ToolNetServiceAssembly_DoorsServiceEngine
INFO  - ServiceUnitLifeCycle      - Initializing service unit: ToolNetServiceAssembly_DoorsAdapter_Out
INFO  - ServiceUnitLifeCycle      - Initializing service unit: ToolNetServiceAssembly_DoorsAdapter_In
INFO  - ServiceUnitLifeCycle      - Initializing service unit: ToolNetServiceAssembly_DoorsServiceEngine_In
INFO  - ServiceUnitLifeCycle      - Initializing service unit: ToolNetServiceAssembly_Installer
INFO  - ServiceUnitLifeCycle      - Starting service unit: ToolNetServiceAssembly_DoorsServiceEngine
INFO  - DoorsServiceEngineConsumerHandler - doStart()
INFO  - ServiceUnitLifeCycle      - Starting service unit: ToolNetServiceAssembly_DoorsAdapter_Out
INFO  - ServiceUnitLifeCycle      - Starting service unit: ToolNetServiceAssembly_DoorsAdapter_In
INFO  - DoorsConsumerListener     - using default port 5094 ❼
INFO  - DoorsConsumerHandler      - ConsumerHandler started.
INFO  - ServiceUnitLifeCycle      - Starting service unit: ToolNetServiceAssembly_DoorsServiceEngine_In
INFO  - ServiceUnitLifeCycle      - Starting service unit: ToolNetServiceAssembly_Installer
INFO  - AutoDeploymentService     - Directory: deploy: Finished installation of archive: ToolNetServiceAssembly.zip
```

During startup of Apache ServiceMix, the JBI environment is set up and the prototype components contained in the ServiceAssembly are started in several steps:

❶ Upon launch, the core ServiceMix configuration is applied, which contains internal settings for Spring configuration, the classpath, JBI container configuration, as well as security and network settings

❷ After ServiceMix configured itself, it starts up the JBI container, which includes the message flow to be used (e.g. ActiveMQ, SEDA or JMS), and initializes the management infrastructure required by the JBI specification, including MBeans for managing the JBI container itself and for installing and starting additional components

❸ When the JMX MBeans provided by ServiceMix are set up, the connector URL is published. Using a JMX management console, users can now connect to ServiceMix using this URL. This is the intended way to interact with the prototype in the use case described in Section 6.4.2.5.

❹ ServiceMix supports restoring the state of a previous run, e.g., after a network failure, Adapter problem or even a server crash. Upon restart, ServiceMix will resend any Messages still in the queue, depending on the message flow used.

❺ After internal and ChainBuilder components (not shown) have been started, the custom `BindingComponent` (*ToolNet-BC-Doors*) and `ServiceEngine` (*ToolNet-SE-Doors*) are installed, started and initialized, as specified in the JBI deployment life cycle in [JBI], *section 6.2.4 "Deployment Life Cycle"*.

❻ Lastly, the ServiceAssembly that realizes the prototype is started and the contained *ServiceUnits* are initialized and started, providing the custom components set up previously with custom configuration as required by the use case (see Section 6.4.4.2)

❼ When the components have finished internal startup procedures such as registering the JBI MessageEndpoint and setting up JMX MBeans, the custom `Consumer` Thread starts its work by registering a `ServerSocket` and listening for input from DOORS.

At this point, interaction with DOORS is possible by using the ToolNet extensions inside DOORS, or by invoking MBean operations on the custom prototype MBeans using JConsole, as explained in Section 6.4.2.4. A graphical representation of the deployed prototype ServiceAssembly when viewed with JConsole is shown in Figure 6.10 below:

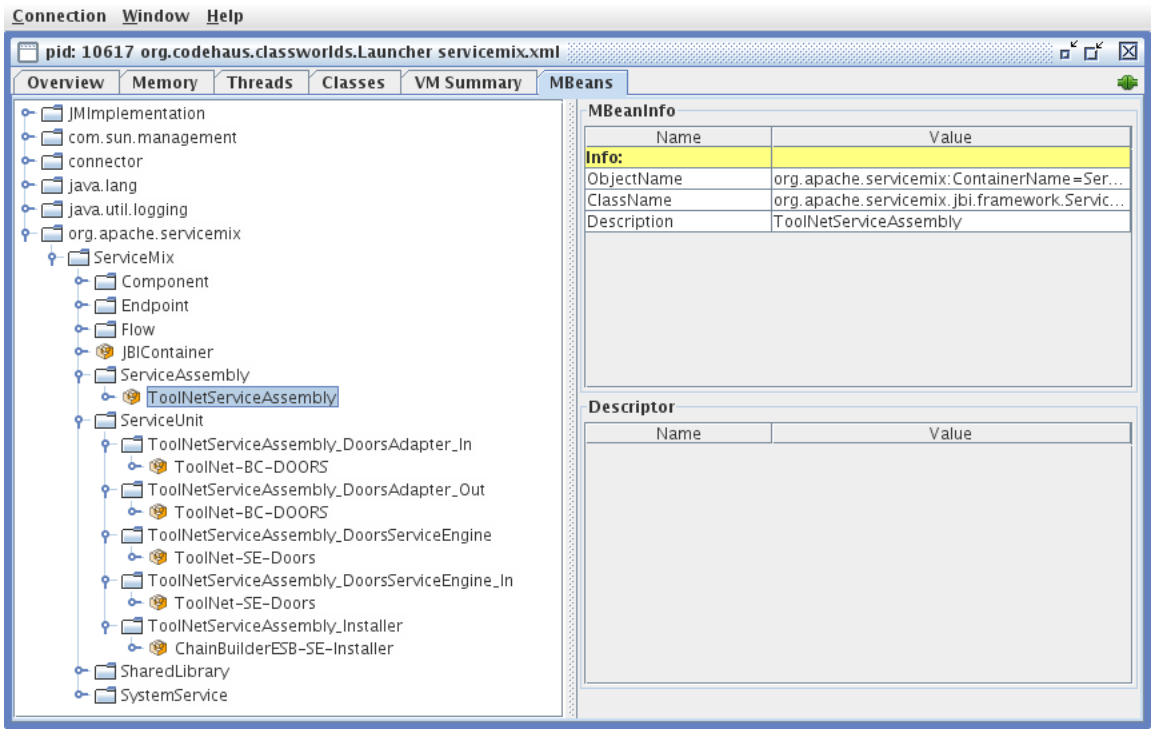


Figure 6.10: Deployment view of the ToolNetServiceAssembly in JConsole

For a full walkthrough accompanied with screenshots showing the complete use case, please refer to Section B.3.

Chapter 7. Critical Evaluation of the Prototype

"When I am working on a problem I never think about beauty I think only of how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."
--R. Buckminster Fuller

7.1. Problems Solved

JBI provides a standards-based, service-oriented integration platform that fits well into the tool integration domain, as has been examined in Section 4.3.2. Although being a Java standard for enterprise integration, JBI does not assume a homogeneous Java landscape, but embraces heterogeneity in systems and software, as often encountered in the enterprise domain and also in typical tool integration scenarios.

By consequently applying JBI concepts throughout the solution, many problems of existing tool integration approaches, esp. ToolNet, have been successfully solved: Where [Mauritz2005:12] identified the demand for a common plugin-standard for integrating tools, stating that "Currently there is no standard available for supporting general 'plugins - tools interaction' and consequently these plug-ins are fully dependent on specific tools.", and suggesting JCA as a possible solution, there is now a standardized way beyond JCA, which was found to bee too limiting in Section 4.2.2.2: in JBI, *BindingComponents* and *ServiceEngines* provide a rich foundation for a standards based, dynamic and extensible tool integration framework, which has been shown in the prototype.

BindingComponents have proven to be a good choice for integrating external tools in a loosely coupled, service-oriented way, using uniform WSDL descriptions to expose tool functionality as services and abstracting from tool-specific interfaces and protocols. Where necessary, BindingComponents allow the usage of native libraries, e.g., using JNA, in a cross-platform way, as shown in the prototype with the *DoorsBindingComponent* (see Section 6.4.2.1). The use of JNA in favor of directly using the low-level JNI layer has proven to be very straightforward and efficient, as no header code has to be generated (and maintained), but only a Java interface that corresponds to the native functionality. This results in a lightweight, homogeneous Adapter design that allows for efficient integration of tools that provide a library interface.

The Adapter-based approach follows the design principle of "integration by encapsulation", as proposed in [Gautier1995] and is also identified as an enterprise integration pattern, the *Channel Adapter*, in [EIP:127].

For higher-level integration above the protocol layer, which was realized with a custom BindingComponent, *ServiceEngines* have been successfully used to interact with tools, e.g., using scripting to integrate DOORS in the prototype. While the prototype does not support configuration of scripting commands, a full solution could use *ServiceUnits* for deploying updated scripts to accommodate changes in tools, e.g., after an update to a new version.

While the JBI standard and associated API is complex, it is based on open service-oriented standards and is very consistent, thus easier to work with than proprietary APIs like ToolNet, where much knowledge of the framework internals is necessary for building Adapters or adding new Services. Recent advances in tool support will make it easier to work with JBI as a component developer and provide more support during all stages of integration development – something which is impossible to achieve with custom built solutions.

The JBI message bus (*NormalizedMessageRouter*) represents a major evolution in handling communication among integrated components, as it standardizes the core concepts of a message bus (c.f. [EIP:137]), but leaves enough room for implementations to choose the concrete topology for realizing the messaging backbone: from

a central enterprise service bus to distributed buses, or more agile peer-to-peer topologies like Grids (see Section 8.5). Existing tool integration solutions are often limited to CORBA middleware, custom solutions, or direct web service-integration, which has been shown to be insufficient in Section 3.3.3.1.

Using normalized messages (c.f. [EIP:352]) provided a convenient way to handle functional and data integration, e.g., for sending commands and associated parameters to integrated tools, and for receiving requests together with data from tools. Command scripts, requests and data were packaged inside the message's XML DOM and the actual tool communication was handled transparently by a custom BindingComponent. Because translation between tool-specific formats and the normalized XML message format is handled by the BindingComponent, tools are loosely coupled to the integration backbone, so other components can simply operate on the XML message, handling data they are interested in (e.g., metadata or attachments).

By consequently using a common, standardized and widely adopted service definition format, WSDL, tools are exposed as a collection of services, which greatly simplifies integration with existing services and allows transparent combination with web services, e.g., for accessing a shared repository (see also Section 8.6 that shows some advanced possibilities that result from this design). WSDL is perfectly aligned with JBI's separation of protocol level vs. application level integration, which greatly supported the goals of the ToolNet architecture redesign, and also helped building the prototype because tooling is very similar to common SOA tools based on web service technologies.

The use of ChainBuilderIDE for developing the prototype and the bundled Apache ServiceMix JBI runtime provided a rich and well integrated solution, both during design time and runtime. The visual integration designer in the ChainBuilderIDE provided a straightforward way to construct the demo scenario (see also Appendix B) including custom components, which were directly available from within the designer. Automatic generation of JBI skeleton code and associated configuration helped in coping with the complexities in developing custom JBI components.

Finally, JMX proved to be a simple but effective way for adding a user interface to the tool integration prototype, with the positive side effect of enabling standards-based manageability of integrated tools. The management interface is not bound to a particular platform or UI toolkit, which is a current limitation of the RCP-based ToolNet Desktop[1], and can be accessed using a web interface or any JMX-compliant client console, such as *MX4J* or *JConsole*[2]. Both can be extended with custom tabs and views, allowing for advanced interfaces tailored to tool integration needs. For end users, the tool's native interface, enriched through Tool Adapters, remains the primary focus of interaction, so the possibilities for extension of current JMX consoles should provide sufficient ways for exposing the integration framework's functionality to users and administrators in a suitable form (e.g., for Service-, Adapter- or Session management).

7.2. Comparing the Prototype to ToolNet

An implementation specific comparison of the two approaches was given in Chapter 6. Table 7.1 shows a high-level comparison of both solutions, illustrating the key characteristics of both solutions in a compact comparison matrix:

Aspect \Solution	ToolNet/Eclipse	Comment	ToolNet/JBI	Comment
Architecture	plugin-based (Eclipse/OSGi)	partially standards-based	fully service-oriented	fully standards-based

[1]Although the ToolNet Desktop is based on a standard UI toolkit, RCP, it is still limited to a specific client technology; only recently, RCP has moved beyond the desktop with eRCP and RAP, but does not provide any management functionality.

[2]the recently released VisualVM [http://marxsoftware.blogspot.com/2008/08/from-jconsole-to-visualvm.html] allows even broader customization and is fully based on plugins, see the migration guide From JConsole to VisualVM [http://marxsoftware.blogspot.com/2008/08/from-jconsole-to-visualvm.html]

Aspect \Solution	ToolNet/Eclipse	Comment	ToolNet/JBI	Comment
Event-Driven	partially	some support for Events, but not very scalable/advanced	full support for EDA including complex event processing	various Event Processors available: IEP, SEDA, …
API	custom API	low reusability, much manual (re)coding needed	JBI defines a standard infrastructure with BindingComponents, ServiceEngines and a message router	loose coupling, clean separation of concerns, high reuse, rich component community and market
Language Platform	Java	fully Java-based, not easy to integrate non-Java Adapters or scripting	Java/Web services	interoperates through Web Services, WSIT, Scripting-ServiceEngine, JSR-223
Data Integration	API-based, via custom Links/Relation-Service	Adapters map ToolObjects to ToolNet-Objects, stores Links in database; Object-mapping not always possible, no updates, custom Object-definitions	high-level data integration using JBI ServiceEngines, low-level data integration through JBI BindingComponents	clean separation between semantic layer and protocol layer, abstracts from data sources, can reuse many JBI components for conversion/mapping[a]
External Bindings	proprietary	fully Tool-dependent and custom built into Adapters	existing or custom BindingComponents or JCA Adapters; some external JBI connectors available (e.g., SAP, CICS, CORBA, JCA)	standardized integration of external systems, custom extensions added as needed (yields high reuse and clean design)
Messaging	proprietary	ToolNet backbone	JBI NormalizedMessageRouter	different implementations available (ActiveMQ, JMS, …) for different interaction styles (simple, SEDA,…)
Adapter Design	heavyweight, Adapter is tightly-coupled to the integration framework	weak separation of ToolSide and ToolNetSide-Adapter, much knowledge about the framework internals, manual support code in Adapters	lightweight, loosely coupled, embraces SOA: WSDL interfaces, XML message format, integration of web standards	location transparency and separation of business logic (in ServiceEngines) from transport-logic (BindingComponents); external functionality transparently exposed as Services

Aspect \Solution	ToolNet/Eclipse	Comment	ToolNet/JBI	Comment
Interface Design	custom API with some Web-Service support	API-centric, not really service-oriented	uses WSDL as the standard component interface	not limited to Web-Services, but uses WSDL as a common interface description schema for all components (also external systems)
Implementation	closed/mixed	using open source or closed/custom APIs as available	open specification using open standards	fully based on SOA standards and concepts, Adapters may need to include proprietary code
Reuse	only with access to Adapter source	not much reuse of Adapter(-service)s, service endpoints called directly	high reuse of Services and Components	Adapters can transparently reuse other Adapter's services, components can be reused (across runtimes)
Relations	supported through custom RelationService	user manually defines Relations, are stored in a database, Adapter has to support Linking functionality	not targeted by JBI, but dynamic lookup of Services is supported	could be implemented as a ServiceEngine that applies the SDO standard[b]
User Interface	custom Eclipse RCP-application	needed to define Relations, controlling Adapters (no lifecycle-management), accessing the Tools	any JMX-based management console; many JBI implementations also provide a rich web interface	no special tools needed, uses existing tools and technologies: JMX enables rich management access for lifecycle-management, Tool and Adapter control; graphical and command line access
Tool Integration	semi-transparent	Tools have to be started from within ToolNet	transparent	Tools connected by Adapter when online
Management Access	standalone ToolNet Desktop, based on Eclipse RCP	uses proprietary management access, cannot be integrated into existing management infrastructure	using JMX	extensible, standards-based management architecture, can be controlled from any JMX interface (JBI specifies required JMX MBeans)

Aspect \Solution	ToolNet/Eclipse	Comment	ToolNet/JBI	Comment
Lifecycle Support	simple/static	Adapter-states defined in API, can be controlled via the Desktop	fully standardized in JBI/dynamic	standard component lifecycle, exposed for management access
Development Support	custom API	complex API with many external dependencies, few documentation	open, standards-based API	API well documented but complex, not easy to write custom components but source code available
Runtime availability	ToolNet (single)	only one runtime available (integrated solution)	any JBI-compliant runtime environment	e.g., ServiceMix, OpenESB, PEtALS, Mule or commercial
Integration Into Existing Systems	none (static, precompiled package)	RCP-application, custom server and custom Adapters, static integration solution	standalone, in JEE app server, connected to message queue (depends on implementation)	supports complex and large scale setups
Web-Services Integration	possible through Adapters	Web-Services not supported natively, but used in some Adapters	native support for web services (WSDL used as service interface in JBI)	web services can be integrated directly
Dynamic Deployment	no	static, precompiled package	yes, fully dynamic	dynamic lifecycle model in JBI supports hot deployment of Adapters and dynamic reconfiguration at runtime
Configuration	static/custom	some Adapters offer configuration using Property-files, no support for configuration of tool-relationships	dynamic/standardized (XML-configuration and container model)	concept of composite applications (ServiceAssemblies), configured via ServiceUnits, supported at runtime
Workflow Support	none	only custom-coded cooperation of Adapters	standards-based (BPEL ServiceEngine, dynamic integration languages)	e.g., ApacheCamel, Rules-engines (see Section 8.3)
Tooling	none	no integration designer or other editors available, only XML editor or Java IDE	any editor that supports the JBI-model and XSL	e.g., Eclipse STP, ChainBuilderIDE, NetBeans, FUSE, …
Distribution	explicit	not transparent, via Proxy interfaces	implicit; support depends on JBI imple-	transparent, often realized with JMS

Aspect \Solution	ToolNet/Eclipse	Comment	ToolNet/JBI	Comment
			mentation (not standardized)	
Routing	partially/custom	some support, but not transparent, has to be hard coded into Adapter	full routing support (part of the specification, as one of the core NM**R** features)	various routing engines available, also support for DSLs (ApacheCamel, IFL[c])

[a]see also Section 8.2

[b]see Section 8.2

[c]see also Section 8.3

Table 7.1: Comparison of the proposed solution with ToolNet

The comparison clearly shows the advantages of the new approach, being standards based and fully based on service-oriented integration using messaging and common WSDL-based interfaces instead of a proprietary API with the need for inheritance and direct method-invocation in Adapters. However, with JBI being a relatively young standard that has never been used for tool integration before, and service-oriented integration still being an emerging field, the new approach is not without its own challenges and limitations, esp. for Adapter developers, which will be examined in the next section.

7.3. Remaining Challenges

Everything is a compromise. That's what you learn. We're always trading off content and date and resources. Nothing we do is ever perfect, because if it was perfect, it would be late, and being late would make it not perfect.
--Bill Shannon, in an interview on Java EE 6

Although JBI 1.0 was released in August 2005, it is only now being adopted by integration vendors and developers, which provide the needed development and tool support. E.g., ChainbuilderIDE was only released in August 2007, 2 years after the specification had been released. JBI has not yet been used outside the enterprise integration domain, even less for a general approach to desktop tool integration. Also JNA, used for communicating with a C library interface in the prototype, is still largely unknown, although being used in some projects successfully and aleady at release 3.0.

Only Apache ServiceMix, the JBI runtime, is already a widely adopted and proven product at version 3.2 (with a release of version 4.0 being imminent). Because it builds on existing, reliable components such as the ActiveMQ messaging backbone, it served as the first reference implementation of the JBI specification and is widely used in various integration products such as IONA FUSE.

Because BindingComponents communicate directly with the tool to be integrated, they are an integral part of the integration framework and directly affect the rest of the system. So, a certain level of quality, stability and performance has to be ensured so as not to compromise other components and subsequently degrading the user experience or even affecting runtime stability. For example, the DOORS API, by default, issues an `exit()`-call when an error is encountered. This resulted the prototype to crash unexpectedly, as the ServiceMix runtime was simply shut down with an error that did not indicate the origin at first, being the DOORS library itself. After some investigation, it showed that the API provides a function, `api_exitOnError(bool)`, for controlling the behaviour in error conditions. By disabling this questionable automatic, the problem was solved and errors could be caught by the Adapter accordingly. It is therefore advisable to perform thorough testing of BindingComponents before including them into a production-level implementation of the proposed solution, as errors or instability in tool interfaces are directly propagated to the Adapter (BindingComponent) that integrates the tool. The following sections shortly cover development issues and another important aspect, quality of service.

7.3.1. Development Complexity and Tool Support

As mentioned before, a major challenge in developing with JBI was the complexity of the API, resulting in a notable initial development cost. The lack of tool and developer support for building custom components adds even more to the initial complexity when starting with JBI development. JBI 1.0 and most tool support is targeted at enterprise integration designers that wish to access disparate services from within business processes, using Adapters available in a prepackaged JBI solution. As a result, most examples and documentation focus on integrating web services, XML transformation, event routing and business process management using existing JBI components.

On the specification level, JBI defines an API for running custom components but not for their implementation, which results in much hand written code although being based on common patterns. While there is source code available for several BindingComponents and ServiceEngines, it cannot be easily reused and often depends on particular API extensions, e.g. *ServiceMix's JBI Component Framework* (JCF)[3] or ChainBuilder's Common Services Layer (CCSL, c.f. [CBESB:44]), that are often not fully JBI compliant (i.e. do not run unmodified in another JBI implementation or introduce additional dependencies like custom libraries)[4]. This was not acceptable for a prototype that should demonstrate a general JBI-based approach to tool integration. A complete example for implementing components with the JBI API is given in [JBIDev], which was used as a basis for the custom component but had to be adapted for the selected ChainBuilder IDE and API.

Another challenging aspect of JBI development is messaging: parsing and creating messages is not trivial, as the XML DOM has to be inspected and manipulated for handling custom message parts; there are no standard methods defined for operating on the message at a higher level, above the data level. Also the construction of a MessageExchange has some pitfalls, which were partially due to the ChainBuilder API, but also due to the complexity of setting up a MessageExchange and implementing required callback methods as expected by the JBI runtime.

Also, depending on the JBI runtime used, debugging custom components can be difficult, as relevant logging information for runtime diagnosis is not always available in the desired detail, e.g., for tracing the message flow through the NormalizedMessageRouter. A simulation of the composite application without involving a real runtime would be very helpful and speed up development, but is not yet available.[5]As a result, developing the BindingComponent or Tool Adapter in general will be the most involving part when realizing the proposed solution on a greater scale[6]. Also, because of the project scope and the complexity with messaging and handling endpoint resolution, the current prototype does not fully explore the dynamic configuration and query possibilities in JBI.

The API complexity and related problems are also identified by Guillaume Nodet, a ServiceMix developer and principal engineer at IONA (now Progress), in his foreword to [Rademakers2008]: “JBI 1.0 has some shortcomings: the JBI packaging and classloader architecture, the mandatory use of XML everywhere in the bus, and the fact that writing a JBI component isn’t easy.”.[7] For the major target group of integration designers however, these shortcomings are acceptable since they do not have to implement custom components and message flows, but can rely on third party support. In the context of this thesis, these challenges were outweighed by the unique service-oriented approach to integration, which allowed to realize a high-level standards based tool integration solution that builds on existing and proven APIs and implementations.

[3] see the web page Apache JBI Component Framework [http://servicemix.apache.org/jbi-component-framework.html]

[4] PEtALS, according to the project homepage, offers API extensions that do not break JBI compatibility, but had no tool support when the prototype was developed.

[5] an Interceptor-based approach, the Message Tracking Aspect Interceptor [http://wiki.open-esb.java.net/Wiki.jsp?page=ProjectFujiAspectInterceptorsOverview], is currently underway for the next version of OpenESB, implementing JBI 2.0 Interceptors, see also Section 8.1

[6] This is also noted in the ServiceMix FAQ for Component Developers [http://servicemix.apache.org/should-i-create-my-own-jbi-components.html]

[7] see also the blog article fun facts to know and tell about JBI [http://coverclock.blogspot.com/2007/01/fun-facts-to-known-and-tell-java.html]

It is expected that most of the shortcomings outlined here will be addressed in JBI 2.0 and with currently emerging second generation tool support, e.g., Eclipse ServiceToolsPlaform (introduced in Section 3.2.4.2), which integrates several integration standards, including SCA and JBI, and common enterprise integration patterns in a unified visual editor with design and runtime development support. In the meantime, documentation and general developer support through communities and Wikis is getting better, and several JBI implementations are preparing for the next major release, such as the imminent release of ServiceMix 4.0 or ChainBuilder 2.0.

7.3.2. Ensuring Quality of Service

The requirements in Section 4.1 include reliability, scalability and security, which was not covered in the previous sections, as it was out of scope for the prototype and is not fully specified by JBI. However, the selected runtime, Apache ServiceMix, and the underlying message queue [ActiveMQ], support many advanced requirements not fully addressed by the JBI specification, but necessary for large-scale deployments and for a production quality tool integration solution.

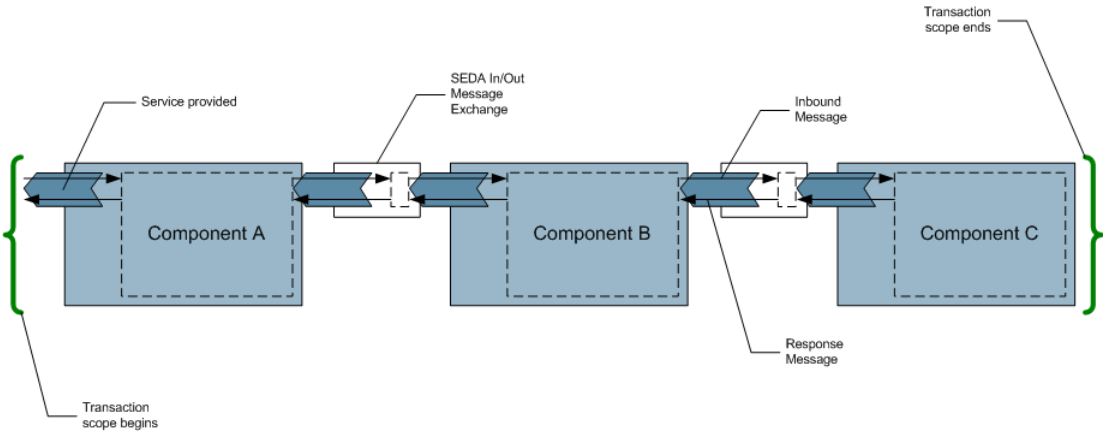
The JBI specification supports manageability through the JMX standard (see Section 4.2.1), allowing lifecycle management and monitoring of components, but also monitoring of the runtime, including used CPU time and memory usage down to thread-level (c.f. [Rademakers2008:386]). By adding custom MBeans for ToolAdapters (i.e. BindingComponents) and higher-level integration services (i.e. ServiceEngines), additional parameters can be exposed for diagnosis and control, such as response time of tool interactions, or the possibility to start and stop Tools, Adapters and Services from a central console.

[JBI:207-214] also proposes a way to handle reliable transactions, ensuring Quality of Service. The specification identifies the following key parameters, which are accompanied with possible solutions using the selected JBI implementation, Apache ServiceMix:

- *reliability:* messages have to be delivered with a certain level of reliability, depending on the solution. For tool integration, the requirement depends on the Service invoked. While a lost `SHOW_OBJECT` request would remain almost unnoticed, users would not be forgiving when a `SAVE` request was not transmitted. ServiceMix offers reliable messaging through the ActiveMQ message queue implementation, which also supports clustering, persistence, and distributed failover (see below)

- *transactions:* composite applications or services may need to share context information when exchanging messages, which is usually handled by using transactions. JBI provides the basis for transactions by defining four *MessageExchangePatterns* (MEPs), but leaves the implementation of related transactional context to NMR implementors. ServiceMix (see the project's page on *ServiceMix Transactions*) offers support for synchronous and asynchronous transactions by using the SEDA or JCA flow, respectively. Figure 7.1 illustrates a synchronous transaction using a SEDA message flow.

  JBI 1.0 does not address distributed transactions, as at that time, it was found that “standards for such transactions are not yet mature enough to be incorporated by JBI directly” [JBI:208]. As mentioned above, ServiceMix does support distributed transactions using ActiveMQ.



(from [ServiceMix], page *Transactions*)
Figure 7.1: Transactions support in Apache ServiceMix

- *persistence:* in the context of JBI, persistence is defined as “the ability to persist the state of a Message Exchange at [a] defined point during the Message Exchange lifetime.”. Persistence of message exchanges is only indirectly supported in ServiceMix to realize recoverability, see below.

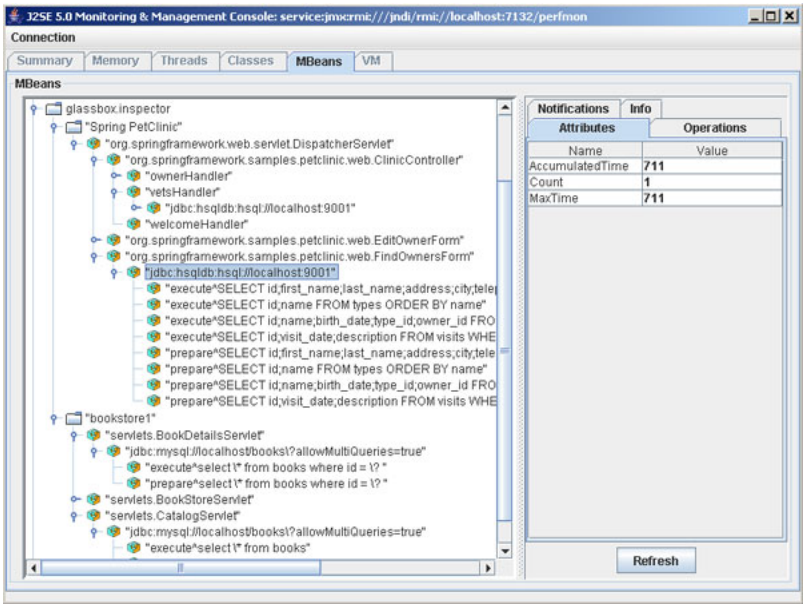
- *recoverability:* in the JBI specification, recoverability is defined as “the ability of an active Message Exchange to be recovered during restart recovery to some consistent point in its lifetime.”. The underlying message queue in ServiceMix, which is ActiveMQ by default, keeps messages in a message store (or a JDBC-enabled database) until they have been successfully delivered to the target (see the *ActiveMQ persistence* page for more information). When the message cannot be sent to its destination because a component is not reachable within the timeout period, or the component or runtime crashes, the message exchange is restored as soon as the target component becomes available again, or when the runtime is restarted. This has been tested successfully in the prototype, when the BindingComponent caused a runtime crash.

- *secrecy:* this aspect ensures “protection of information from being disclosed to outside parties”, and is left to JBI implementations as it only affects storage of information outside the NMR. As noted in Section 4.3.2, ServiceMix supports core security concepts such as authentication, authorization and message encryption, which makes the proposed solution applicable to scenarios where confidential data is shared between tools (e.g., licensing information, login information for tools, company confidential information), even in distributed settings. [Rademakers2008:272] explains how to implement security with ServiceMix, using the WS-Security web standard.

To summarize, most of the manageability and quality of service-features can be satisfied with Apache ServiceMix, including support for high availability and clustering[8].

Sometimes however, more formal processes need to be satisfied, such as *Service Level Agreements* (SLAs). [Glassbox] is an open source solution that adds support for service-level management and monitoring, integrating with JMX and utilizing aspect-oriented programming concepts[9]. The Glassbox container is installed into the application server's directory and runs in the same JVM as the components to be monitored. Using aspect-oriented programming (AOP), all transactions are monitored for several fault patterns, which can be adjusted to monitor SLA violations, as shown in Figure 7.2 below:

[8] see the ServiceMix page on Clustering [http://servicemix.apache.org/clustering.html]
[9] see the article Glassbox: How it works [http://www.glassbox.com/glassbox/HowItWorks.html]



(from [Glassbox])
Figure 7.2: Service Monitoring with Glassbox

ChainBuilderESB [CBESB:33] supports QoS monitoring and SLA enforcement by monitoring for user-definable alert conditions (e.g., resonse time or error count), logging alerts to a database, and resending messages as necessary[10].

Also, external monitoring with existing service administration systems such as Nagios is possible[11], other management consoles such as HP OpenView or web based solutions such as Zenoss are supported via the normal JMX management layer. In comparison to tightly integrated solutions such as Glassbox, these systems provide only coarse-grained monitoring, as they have no access to internal information about the JBI runtime or the JVM itself, thus they can only track defects explicitly exposed for management by JBI components.

On a more general level, WSLA [Keller2003] is a framework developed by IBM research, which specifies a language based on XML schema and an associated runtime for monitoring service-level agreements in a web services environment. The framework is applied in [Fung2005], who extends BPEL4WS (which has now become WS-BPEL) with attributes to support QoS metrics, and integrates the standard with WSLA. [Nepal2008] adds extensions to WSLA for coping with collaboration among multiple parties, and proposes WSLA+ as a result.

Related concepts such as business activity monitoring are supported through Apache Camel, which is shortly introduced in Section 8.3.

7.4. A Migration Scenario for ToolNet

A migration concept for the current ToolNet implementation was defined as a goal of the prototype in Section 6.2.5. A conceptual mapping between the existing DoorsAdapter and the new BindingComponent used in the prototype has been provided in Table 6.2 before. A general comparison of the concepts follows inTable 7.2 below:

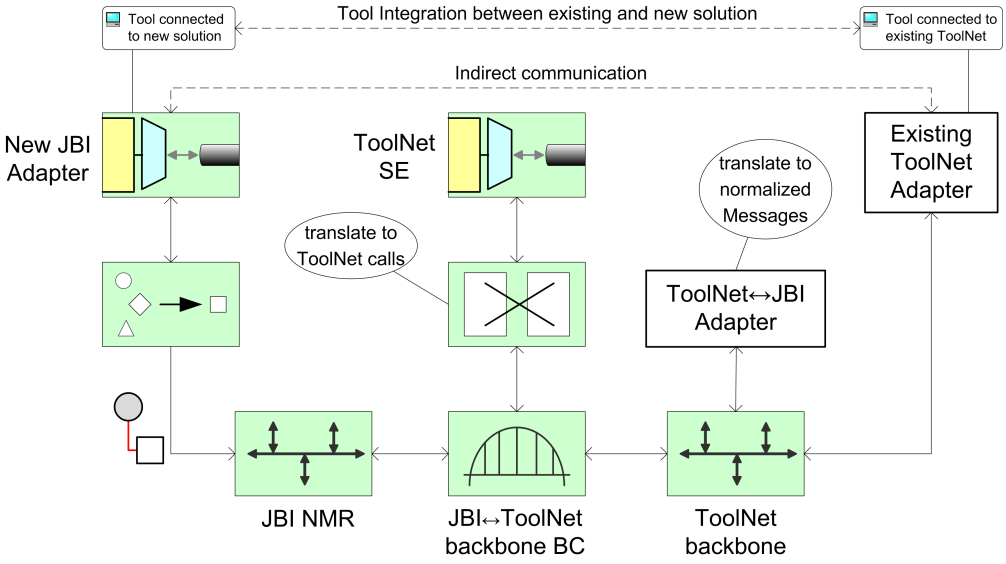
[10]see also the blog entry by Eric Lu, CTO Bostech Corporation, on Why choose ChainBuilder ESB over other Open Source ESBs? [http://chainforge.net/pLog/index.php?op=ViewArticle&articleId=21&blogId=1]
[11]see the article ActiveMQ stomp end to end test for Nagios [http://just-another.net/2008/09/03/activemq-stomp-end-end-test-nagios/]

ToolNet	ToolNet/JBI	Migration Strategy
ToolSideAdapter (integration with Tool interface)	— (no modification required)	Tool-side extensions can be reused as is
ToolSideAdapter (connection to ToolNet)	BindingComponent	has to be retrofitted into a JBI BindingComponent, Tool-side communication code can be reused
ToolNet Services	ServiceEngines	have to be transformed into ServiceEngines and exposed via WSDL
API calls	normalized messages	JBI is message-based, not method-based: API has to be redesigned to be service-oriented
Extension Points (OSGi/Eclipse)	Endpoints (Bindings)	as part of component migration
ToolNet backbone	— (obsoleted by NormalizedMessageRouter)	as soon as Adapters are migrated, the ToolNet backbone is no longer needed; until then, a BindingComponent can be used as a bridge

Table 7.2: Mapping ToolNet Concepts to JBI Counterparts

As mentioned in Section 5.3.7, ToolNet already provides WSDL definitions for some of the framework components and for Adapters that integrate with Web services or with Tools that provide a WSDL interface (e.g. ToolNet's Word 2003 Adapter uses web service integration). As these endpoint definitions are already based on WSDL, they could be refactored with relative little effort into JBI *ServiceEngines* (in case of ToolNet Services) or BindingComponents (for Adapters).

For a smooth migration path, or If some of the existing ToolNet components cannot or should not be changed, the two message buses could be bridged and two new components would realize API-level translation of service calls, one on the ToolNet side and the other on the JBI side, as illustrated in Figure 7.3, using familiar enterprise integration pattern icons from [EIP]: a BindingComponent connects the existing ToolNet backbone with the JBI NMR by communicating with a corresponding Adapter on the ToolNet side, following the message bridge-pattern [EIP:133]. The BindingComponent performs protocol-level translation between JBI's normalized messages and ToolNet service requests, notification events or results returned by ToolNet Adapters. A new ToolNet ServiceEngine performs API-level integration between the new Adapters on the JBI side and existing Adapters or Services on the ToolNet side, acting as a mediator that propagates relevant service calls to the ToolNet backbone and forwards ToolNet service calls to corresponding JBI Adapters. This would allow transparently calling existing ToolNet Services or Adapters from the new solution. On the ToolNet side, a new Adapter is introduced that wraps ToolNet service requests and events into remote calls to the new BindingComponent on the JBI side, acting as a mediator in the same manner as the JBI SE, thus allowing ToolNet Services and other Adapters to access JBI Services in a transparent way.



Integrating ToolNet with the new JBI solution: the desired integration between existing and new Adapters is indicated by the dashed horizontal line on the top, while the physical integration is shown below
Figure 7.3: Integrating existing ToolNet components with the new solution

Subsequently, core ToolNet services and functionality could be translated into the new architecture: ToolLinks used in ToolNet for linking data elements in tool models could be realized by a *RelationServiceEngine* that provides Services for managing and querying relations between tools, in the same way as done by the current ToolNet RelationService, but with the added advantage of location and protocol transparency, which means that also web services or other service-enabled systems could easily be extended with relational capabilities.

In a similar manner, Project(Session)s used for managing project-specific collaborative workflows could be realized by a *ProjectServiceEngine*, with the added benefit that Sessions could be easily made persistent and restored in a subsequent session. A possible way to achieve this would be to implement custom `MBean.load()` and `save()` methods that are invoked on startup and shutdown of the component, respectively. In the `save()` method, Sessions are persisted to a database, and on `load()`, the Session data is restored and the Session is initialized accordingly. This happens transparently inside the ProjectServiceEngine, so there is no need for other components to cater for Session initialization or storage.

Lastly, the ToolNet Desktop could be replaced by JConsole, as noted earlier in Section 7.1. All ToolNet/JBI components and their custom attributes and methods are automatically exposed for management by any JBI-compliant runtime, such as Apache ServiceMix. Custom dashboards or other UI (e.g., the relation viewer) could be added by implementing suitable plugins, as shown by several examples on the JConsole (now VisualVM) homepage.

As ToolNet is currently in the process of migrating the underlying component model to OSGi, it is already aligning with open standards and also with JBI, which will use OSGi as its component model in version 2.0 of the specification. Moving right to JBI now would therefore represent a more effective migration path, since manual migration to OSGi would not be necessary, and exchanging only the component layer would still lack the additional high-level advantages and integration facilities of the JBI solution (see also Section 3.2.3.1).

Part III. The Future of Integration: Outlook and Conclusion

Table of Contents

The final part provides insights into emerging and future trends in integration, ending with a conclusion that reflects on the findings in this work.

Chapter 8. Outlook and Further Work

“There's a better way to do it. Find it.”
--Thomas A. Edison

Integration is a vast domain, and only a small part could been shown in this work and in the prototype. While working on this thesis, development has not stopped – JBI implementations and tooling have improved, new possibilities in integration have emerged and even another integration standard was born, SCA (shortly introduced in Section 4.2.2.3). This chapter will look at emerging integration standards, beginning with the next major release of JBI, and shortly point out relevant developments not covered in this work, because they were out of scope for the proposed solution, such as advanced data integration with SDO or dynamic scripting and DSLs for integration. We will then look at the bigger picture in tool integration, beyond software.

8.1. The Future of JBI

While JBI has been successfully used to realize the proposed solution in a prototype scenario (see Chapter 6), as analyzed in the previous chapter, several challenges and open issues were identified in Section 7.3, which is understandable since the standard tries to cover a large and complex field and is only at version 1.0. The specification acknowledges some room for improvement and provides a look at prospective advances in version 2.0 [JBI:205], including the following key areas, which are shortly examined for applicability to tool integration and accompanied by examples of current non-standard extensions in JBI implementations that try to fill the gaps in JBI 1.0:

- *J2ME support:* An embedded JBI implementation would allow mobile access to integrated tools, e.g., remote control of repositories or acknowledgment of long-lasting tool operations. No implementations of this kind are currently available, but several signs indicate the feasibility of embedded JBI, especially since JBI 2.0 will be based on OSGi, which has its origin in the embedded space.

- *custom message exchange patterns:* this would allow more complex conversation patterns and could also be used to integrate human tasks, i.e., activities carried out by people (see Section 8.6 below), but could be realized alternatively with custom workflows like BPEL processes or using dynamic routing languages (see next section)

- *long-lived message exchanges:* currently, message exchange is not optimized for memory footprint, which could be a problem for long-running message exchanges, which would effect, e.g., long running business processes or transactions, or long running tool functions that are called synchronously

- *persistent message exchanges:* persistence on shutdown or failure is already supported by, e.g., Apache ServiceMix, but a standardized way to persist messages and message exchanges would be desirable

- *API improvements* and advanced support for shared libraries: this would remedy a major point of critique, as also identified in Section 7.3, and is a prerequisite for efficient and dynamic development of Tool Adapters, the main component in any tool integration framework

- *distribution:* this was early identified as an important but missing feature of JBI. Nevertheless, many JBI implementations already do support distribution, such as Apache ServiceMix or PEtALS. As distribution is not part of the standard, individual JBI runtimes may chose different ways to implement distribution, making it impossible to mix runtimes or to rely on distribution when developing components.

- *handlers* (now called *interceptors*): these would allow for unobtrusive filtering of messages before they reach the target component, allowing for dynamically adding functionality as needed in a more aspect-oriented man-

ner, without changing the component itself; possible uses include security, auditing and logging, transactions, compression (useful for long-lived message exchanges above) or policies (see below)

- *policy support:*: allowing for specification of required and provided capabilities, ensuring Quality of Service to monitor service-level agreements (introduced in Section 7.3.2)
- *security:* the specification should require advanced support for security in implementations, reusing existing standards such as JAAS or SAML.

In the meantime, JBI 2.0 has been approved as *JSR-312* [JBI2], but not much public information is yet available except for a few presentations (e.g., [Walker2007] who was the co-spec lead of JBI 1.0) and informal blog entries. From these sources, the following main goals for JBI 2.0 can be deduced in addition to the goals outlined above:

- *utilization of OSGi* as the underlying component model, solving component dependencies, class loading issues and providing *service versioning*
- *clearer alignment with SCA*, standardizing the deployment of SCA artifacts in JBI runtimes
- *re-organizing the specification* for multiple audiences, e.g., component developers, integration designers, JBI runtime implementors
- *support for POJOs* (normal Java objects), easing component development: many runtimes support this but in a non-standard way, rendering such components incompatible to the JBI specification and limiting reuse across runtimes
- *better runtime and configuration management* : extension of the JMX management layer, presumably utilizing new features in JMX 2.0, and providing an easier installation method for components, perhaps like Maven or Debian's APT (automatic retrieval and installation of components, including resolving dependencies) – this has been suggested in [O'Neill2007]; also, Apache ServiceMix provides hot-deployment through the filesystem, whereas the Spring framework recently introduced a repository for OSGi components [SpringRepository], which is closer to the APT analogy.
- *less web services/WSDL dependency:* full dependency on WSDL has proven to be problematic in certain situations, as the web services metaphor cannot be easily mapped to all systems, and WSDL documents tend to be complex in structure – an alternative model is proposed in [WSPER2007], which defines a framework consisting of a metamodel and programming language for developing composite applications, building on WSDL, SCA and BPEL.
- *less message normalization:* for improving performance, esp. among closely related components, where the normalization step could be left out (like it is possible with Mule, see Section 4.4.2.3)
- *better support for tooling:* with additional hooks from runtimes, support for debugging could be realized, see also the next point
- *diagramming support:* similar to how SCA defines a UML model for composite applications [SCAUML].

Some proposed features of JBI 2.0 are already partially supported in upcoming versions of current JBI implementations, such as [ProjectFuji] (codename of version 3 of OpenESB, the JBI reference implementation), which is based on OSGi and will support several *Interceptors*, e.g., for simulation of message exchanges. Apache ServiceMix 4 is already available in milestone 1 and is completely based on an OSGi-based runtime kernel, using [ApacheFelix] as the underlying OSGi implementation.

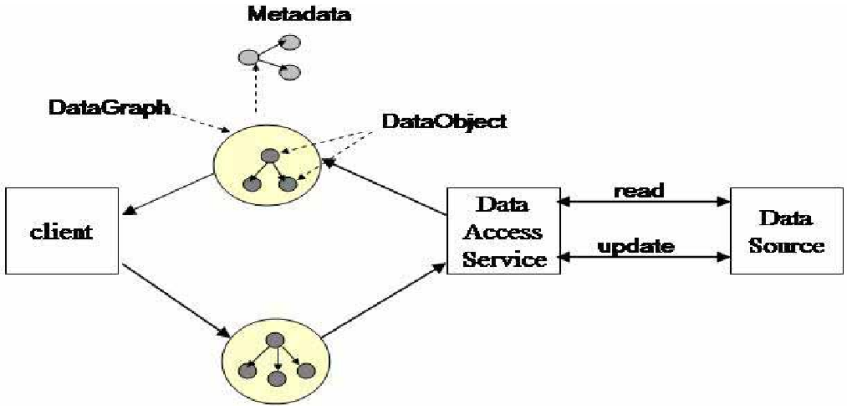
In the tooling landscape, Eclipse STP is quickly becoming an integrated meta-modeling tool for service-oriented integration, providing support for the SCA composite applications and increasingly also JBI, albeit only as a runtime at the moment. IONA provides design-time support for JBI with the FUSE Integration Designer, which

is based on Eclipse STP. Another solution based on Eclipse is ChainBuilderIDE, which will move to version 2.0 soon and was used for the prototype realization. Sun's NetBeans IDE provides excellent integrated tooling support for JBI development. from design to deployment, but focuses more on the design of composite applications that reuse existing components. Support for developing custom components is currently being improved (available with version 6) by providing suitable project types and configuration wizards.

8.2. Future Trends in Data Integration: SDO

According to [SDO2007a], “Service Data Objects (SDO) are designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems.”

The SDO specification comprises several parts: SDO *Core* defines an architecture for high-level data access, abstracting from data sources and representing data objects using disconnected data *graphs*. Data can be accessed and manipulated in various ways, including *XPath* expressions, even when data sources are not available. When the data source comes online again, data can be synchronized when updating or storing data, including efficient change propagation across services. A rich metadata API allows to store and query additional properties (even DataObjects) with DataObjects. The data model is illustrated in Figure 8.1 below:



(from [SDO2007b])
Figure 8.1: SDO's abstract data model

The SDO standard also specifies runtime implementations for all major languages, including Java, C++, PHP and others (see relevant specifications at [SDO2007a]). With Apache Tuscany [Tuscany2008], an open source SDO (and SCA) implementation for Java is available.

Accessing and managing data from heterogeneous sources is one of the key problems in tool integration (c.f. [Gorton2003]), but a closer examination and inclusion into the proposed solution was out of scope for this work. The SDO specification represents a standards-based effort in data integration that would be very well suited for tool integration, and the proposed solution could be extended to utilize SDO for rich data integration. As a case study, Xcalia (see Section 3.3.3.3.2) uses SDO for integrating external data sources. Also, the *Virtual Data Access* mentioned in [SDO2007b:10] can be viewed as a standards-based implementation of the *Virtual Object Space* (VOS) used in ToolNet (c.f. [Geissler2001]).

Although the SDO specification is often related to the SCA standard, because the two standards align well, both can be used independently. SDO could be supported in JBI as a *ServiceEngine*[1], although that has not yet

[1]see the answer Re: Does JBI support SDO [http://osdir.com/ml/java.servicemix.user/2006-06/msg00070.html] from ServiceMix developer Guillaume Nodet on the ServiceMix mailing list

happened. The development teams of Apache Tuscany and ServiceMix are working together to facilitate reuse of SCA composites in JBI runtimes, and SDO would also fit this model very well. Tooling for SDO is provided by [EclipseLink2008] as part of the Eclipse project. *EclipseLink* is an open source persistence framework based on SDO and JPA (*Java Persistence Architecture*, a Java-specific persistence standard), which can be embedded as a set of OSGi services. This would facilitate inclusion into a JBI 2.0 runtime, which is also based on OSGi (see previous section), forming a coherent, standards-based integration solution.

To conclude, *SDO/Java* could be used together with JBI to realize dynamic data integration with support for incremental updates, consistency and transparent, disconnected synchronization (i.e., when a tool goes offline and later comes online), and would be worth further investigation for inclusion into a future prototype.

8.3. Scripting and Emerging Integration Languages

In the prototype, only one Adapter has been realized, using the proprietary DOORS scripting language DXL for invoking tool functions. Other tools may provide a standardized scripting interface using Python, Ruby or PHP, for which Java bindings or ports exist, e.g. Jython, JRuby or Quercus [Quercus] (or the php/Java bridge [pjb]), respectively.

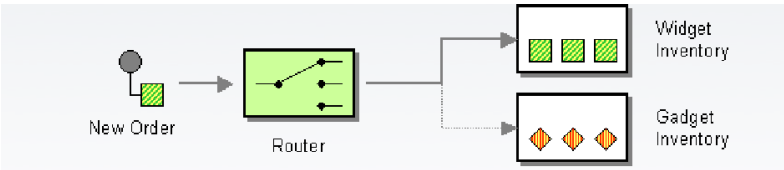
As mentioned in Section 3.2.1.2, [JSR223] provides a standard API for integrating scripting languages into Java applications (part of Java6). The standard is already integrated into JBI implementations as a ServiceEngine, e.g. [ServiceMixScript2008] or [OpenESBScriptingSE], which makes it possible to call a script in a message flow and get the return value of the execution for further processing.

The *DaVinci VM* [JSR292] moves support for scripting languages directly into the virtual machine, resulting in improved performance and transparency in combining scripting languages with Java. The new VM is currently developed as part of the OpenJDK in [DaVinciVM2008].

Generally, there is currently a renaissance of dynamic programming or *language oriented programming* (LOP, c.f. [Dmitriev2004]), which is realized through domain-specific languages (DSLs), introduced in Section 3.2.2. This allows for easier and more lightweight service composition and configuration of composite applications than using conventional, imperative programming languages such as Java. Other advantages include easier support for tooling and validation of integration solutions.

This programming paradigm may be implemented as a routing engine, such as the open source Apache Camel [ApacheCamel], which supports classic enterprise integration patterns (see Figure 8.2)[2]. Camel provides an API and a Java-based DSL for describing typical enterprise integration scenarios like composite applications in an ESB.

[2] see Apache Camel DSL support [http://activemq.apache.org/camel/dsl.html] for an example




```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice().when(header("foo")
            .isEqualTo("bar")).to("seda:b")
          .when(header("foo").isEqualTo("cheese"))
            .to("seda:c").otherwise().to("seda:d");
    }
};
```

(from [Snyder2007])
Figure 8.2: DSL-based routing configuration with ApacheCamel

As part of Project Fuji (OpenESB v3), a more abstract DSL is being developed for specifying the message flow in a composite application, the *Integrated Flow Language* (IFL) [IFL2008], which is still under active development and motivated as thus: “Starting with a rapid, top-down development language, IFL (Integration Flow Language), developers can quickly and easily generate composite applications using a domain-specific grammar.” (from the project page). An example is provided in Example 8.1 below:

Example 8.1: Creating a sample composite application with IFL

```
rss "cnnfeed"
jruby "filter"
xmpp "IM"
file "archive"

route do
   from "cnnfeed"
   to "filter"
   broadcast do
      route to "IM"
      route to "archive"
   end
end
```

(from [IFL2008])

Spring Integration (see Section 3.2.3.2) provides another way to describe the configuration of integrated applications, building on the wide-spread Spring XML-schema and the consistent use of *dependency injection* (ibid.).

For describing more complex runtime logic or for implementing expert systems, rule-based systems like [Drools] offer an efficient, lightweight approach: Drools allows defining several rules that are evaluated dynamically at runtime, which is an implementation of the *Dynamic Router*-pattern [EIP:243]. Separating application logic from data results in better reusability and facilitates more visual tool support than forcing implementation of hard-coded, implicit rules in application code.

While these concepts would be an excellent choice for further improving reusability and agility in tool integration, these languages and frameworks are still rapidly evolving and largely proprietary. Although some implementations comply to standards like [JSR94], which specifies a common API for rule engines, there is still no standard for defining a common rules language itself. As a result, many different and overlapping languages are

currently available, which are bound to specific implementations, hindering reuse and interoperability. In order to fill this gap, a W3C working group proposes a standard for a Rules Interchange Format (RIF) in [RIFWG2008], which is in public review at the time of writing.

8.4. Interoperability with the Non-Java World

Although this work focuses on Java and proposes a Java standard (JBI) for tool integration, this does not mean the solution is tied to the Java platform and only Java Adapters or systems can be integrated. The choice of Java and JBI were in part motivated by the goals set out for the ToolNet redesign, with ToolNet being implemented in Java. This made JBI a good fit and is also an advantage for a possible migration (see Section 7.4). Also, the Java world provides many open source solutions that could be used for realizing the proposed solution, and there is no comparable standard available for other platforms such as .NET. The only alternative is the language-neutral SCA standard, but also there, the Java implementation is currently the most mature and widely used.

For client-side integration of proprietary tool interfaces, *JNA* has been successfully used from within a custom BindingComponent, as shown in Section 6.2.4. This makes it easy to bridge from Java to native or legacy languages, as long as a library interface is provided. When a .NET-based client interface is available, the commercial solution JNBridgePro [JNBridgePro2008] provides tooling (for Eclipse, VisualStudio or stand-alone) and code generation for transparently communicating with .NET applications from within Java applications and vice versa. After selecting the classes that need to be accessed in the target application, JNBridgePro automatically generates relevant proxies that can be used in the client application. This approach is comparable to JNA but works at a more abstract level and also provides automation and visual tooling. In a similar way, Codemesh [Codemesh2006] provides solutions for interoperability with .NET (through JuggerNET), C/C++ (JunC++ion), and CORBA, and also allows C/C++ and .NET clients to access JMS message queues (through JMS Courier).

Message-based integration is still a viable solution for cross-platform interoperability, when distributed integration is needed but service-oriented integration is not applicable. In addition to the commercial solutions mentioned before, the ActiveMQ message queue (which is also used in ServiceMix) also supports clients written in other languages and platforms, such as .NET[3]

. On the server side, Sun is working together with Microsoft to ensure web services interoperability of second generation Web Services, under the umbrella of the *Web Services Interoperability Technologies* [WSIT], a set of Java integration technologies that “enable interoperability between the Java platform and Windows Communication Foundation (WCF) (aka Indigo).” (from the project page). On the Java side, the Metro stack implements several of the WS-* standards (introduced in Section 3.3.7.2), whereas on the Windows side, .NET's WCF performs this role. WSIT also conforms to the *WS-I* standards, ensuring high-level web service interoperability. To use WSIT functionality, no runtime API and hence no code modification is required, but only a configuration file which can be automatically generated by IDEs like NetBeans.

WSIT is an example for service-oriented integration of heterogeneous platforms (using web services). While targeted at web service-integration, WSIT can also be used for tool integration on the desktop, as shown in [Carr2007], where MS Excel is connected to a JEE web service to communicate with a backend storage system. [Neward2007] provides a thorough overview of .NET and Java interoperability, including sample code and more details on the use case involving client integration with Microsoft Excel 2007.

8.5. REST and Resource Oriented Architecture

A tool integration framework should introduce only a thin, lightweight layer that provides dynamic, *ad hoc* combination of tools and facilitates rich communication among integrated tools. Although JBI is very scalable

[3]see the page ActiveMQ Cross-Language Clients [http://activemq.apache.org/cross-language-clients.html] and the article Messaging with .NET and ActiveMQ [http://remark.wordpress.com/articles/messaging-with-net-and-activemq/] for an example

(see Section 8.1 above), a full service-oriented software stack often introduces some overhead, resulting in less transparent operation and degraded user experience.

A new paradigm is currently emerging, *resource oriented architecture* (or *web oriented architecture* (WOA), *resource oriented computing* (ROC)), where the distinction between services and data sources is blurred, allowing a uniform and direct access of mixed resources, as common in enterprise settings (and again tool integration).

> Resource-Oriented Computing solves system and application integration issues by leveraging ESB, domain-specific languages, and shared memory mechanisms for integrating coupling points, not the applications themselves, by promoting event-driven interactions between system components, and by creating logical mappings of resources such as data or computations that are abstracted from the physical manifestation of the system deployment.
>
> —Eugene Ciurana in [Ciurana2008]

The key idea behind this trend is to take successful integration strategies from the web, which faces similar challenges regarding distributed and heterogeneous systems, and apply them to application integration.

Accessing components is possible by simply accessing an URI, in the same way as resources are accessed on the web. By abstracting from the protocol and operating on logical endpoints instead of physical endpoints, a much more dynamic and stable integration can be realized. This concept is not too different from JBI's abstract integration model, but JBI introduces a considerable overhead with its WSDL-centric endpoint description, service assembly configuration and normalized message exchanges. In a resource-oriented way, tools, their operations and data could be viewed as resources and accessed in a simple way, as demonstrated in Example 8.2:

Example 8.2: A possible tool endpoint description in URI-notation

```
toolnet://research.eads.de/doors/highlight/requirement-881
```

ROC can be seen as a generalization of REST [Fielding2000], as it shares the same design principles but provides a more abstract way for accessing resources, adapting to dynamic integration needs. ROC is not bound to `HTTP` for transmitting requests, or a DNS server for resolving endpoints. This enables a richer vocabulary than the few predefined verbs in `HTTP` (GET, POST, PUT, DELETE, …), which is essential for integrating existing systems.

There are several ways to implementing a ROC architecture. IBM is currently adopting REST for implementing application mashups in WebSphere with ProjectZero[4], and a general Java framework for REST-based applications is available at RESTlet.org[5].

For massively distributed systems and large-scale deployments, a Grid topology is a possible solution, e.g. [GridGain] (open source) or [GigaSpaces] (free community version available, restricted to a single node). A spaces-based open source Java ESB using JBI is currently in progress with the [Anageda] project.

Looking at existing JBI implementations, [OpenESB2008] currently adds REST support to the OpenESB JBI implementation, with the goal to “enable OpenESB components to consume/provide web services other than using SOAP/XML. [The] ability to interact REST fully allows OpenESB components to leverage a variety of web services such as Google Apps, Amazon WS, salesforce.com, etc.”. REST is also supported in ServiceMix through REST POJOs[6], building on ActiveMQ's REST support[7]

This would enable integration of existing tools into a new, resource-oriented tool integration architecture.

[4] http://www.projectzero.org/
[5] http://www.restlet.org/
[6] see Apache ServiceMix page on REST POJOs [http://servicemix.apache.org/rest-pojos.html]
[7] see the article ActiveMQ and REST [http://p-st.blogspot.com/2007/12/activemq-and-rest.html] for an example

Users increasingly utilize web based applications as part of their work, and many services nowadays offer a REST interface (e.g., Amazon, GoogleApps, eBay). However, these are isolated applications that incur similar problems as encountered in desktop integration, with isolated and proprietary APIs that bind data to the original application and make reuse and combination difficult, as noted earlier in [Burcham2005]: “While these web applications are manipulating domain-specific information they are doing precious little to expose that information in interoperable form.”. The article proposes a “web clipboard”, where information can be easily shared among web applications just like between desktop applications. This is however still a low-level form of integration and requires manual intervention, but high-level information sharing is hindered by several other obstacles, such as authentication or proprietary APIs. This need has been addressed in part by introducing a common authentication scheme, [OpenID], and by proposing common domain-specific APIs, such as Google OpenSocial API [OpenSocial] and Amazon A9 [OpenSearch].

By integrating web applications into a common tool integration framework using emerging APIs and architectures, a rich and location-transparent end user experience could be provided, reusing existing online resources but adding semantic integration, and combining online with offline applications, thus following the ongoing *convergence* of desktop and web based applications. [Chen2007b] proposes such a collaborative, resource-oriented environment, termed the *universal virtual workspace* (UVW), integrating an existing application, Matlab, in a service-oriented integration framework but utilizing resource-oriented concepts.

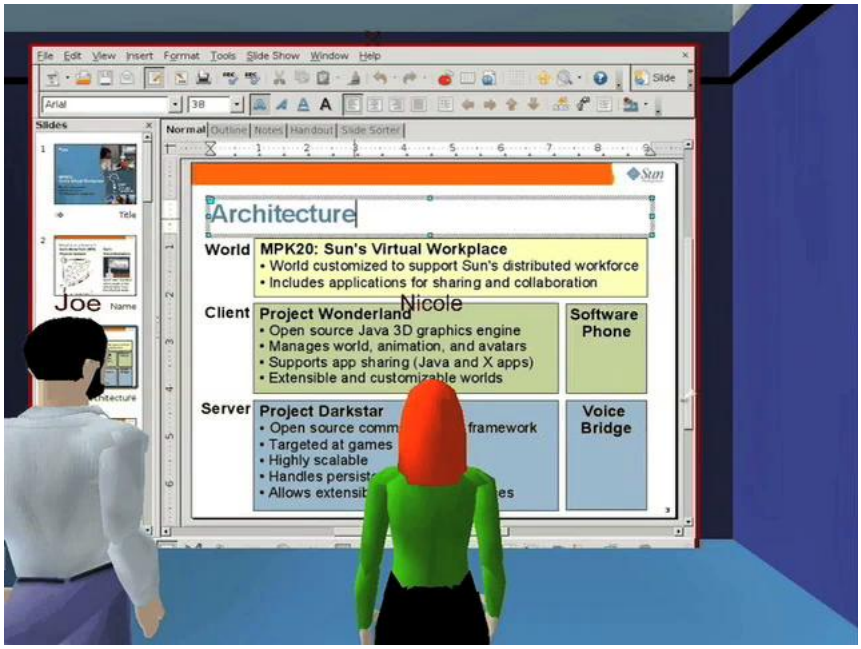
This trend is also reflected in the conventional software development landscape, most prominently the Eclipse IDE, which is increasingly embracing web based development through sub projects like RAP (see Section 3.2.4.2) and the work on the next major release in the Eclipse E4 project [EclipseE4].

8.6. Beyond Tool Integration

A successful tool integration solution must also address the needs of people working with tools. Recent web service standards like WS-HumanTask [WS-BPEL2007] allow to model the coordination of tasks performed by humans in combination with web services. The WS-BPEL extension BPEL4PPL (ibid.), which was recently submitted to the OASIS standards consortium, applies WS-HumanTask to address the integration of human activities into automated business processes. These standards could be applied to tool integration for realizing complex, semi-automatic workflows in a task-oriented way, in combination with employing specialized tools, e.g., for building a complex system in a distributed team. Participants of such a workflow could then collaborate more efficiently, with artifacts shared between tools and authors, as dependencies between people, tools (regarding provided functionality) and data is modeled in a clear and easy-to-follow manner.

The collaborative nature of Web 2.0 – and recently Enterprise 2.0, which can be seen as an adoption of Web 2.0 to enterprise needs, such as security or business functionality – shows the potential of user-controlled recomposition of existing services, assets and information to form new, composite services that serve a specific and spontaneous need. The result is a highly dynamic and rapid workflow that improves problem solving and information sharing, avoiding duplication and isolation of data or functionality. This spontaneous reuse and composition of existing assets is a prime motivation in tool integration and a good solution should therefore strive to apply proven techniques and concepts, such as the consequent use of open standards, to create a flexible and dynamic platform for integrating tools in a user-friendly and user-controlled way.

Project Wonderland [Slott2008] provides an early look at the possibilities of visualizing collaboration among people that work together with different, existing tools on common artifacts shared in a 3d world, as illustrated in Figure 8.3 below:



from the article *Project Wonderland Go ahead, make a scene.*[8]
Figure 8.3: Application sharing in Sun's Project Wonderland

This standards-based open source 3d framework shows the potential of collaborative work in the Web 2.0 era, building on the foundation of composite services. A precondition for this novel collaboration experience however is that tools in the real world are integrated into a loosely coupled extensible system, e.g., using OpenESB[9]. Just as Web 2.0 is a collaborative platform for web applications, *Tool Integration 2.0* could be a new, dynamic and collaborative platform for integrating tools in a user-centric environment, be it the desktop, the web, or whatever the future will bring.

In a more automated manner, such an “integrated tools environment” could automatically combine tools based on current user demands, as envisioned in [Grigonis2008] on IBM's secure mashup technology, called “SMashup”: “The best mashup would resemble a biological organism, a sort of shape-shifting chimera – the user would simply specify the kind of application he or she needed, and the components would intelligently figure out among themselves how to assemble themselves in a way that satisfies the need.”

[8] http://research.sun.com/spotlight/2008/2008-08-19_project_wonderland.html
[9] as shown in this blog entry on OpenESB in Wonderland [http://blogs.sun.com/jason/entry/openesb_in_wonderland]

Chapter 9. Conclusion

With the increasing proliferation of JBI and other integration standards such as SCA, there is now a viable open integration market evolving, with rich design-time support through integration designers and runtime support through an open market for integration components and runtime implementations. This facilitates the realization of solutions that focus on the core task of tool integration, providing additional functionality and flexibility to end users and integrating at the semantic level. There is no need anymore to build custom APIs or to use legacy architectures such as CORBA, resulting in proprietary and complex communication backends that impede development of light-weight and dynamic tool integration solutions, causing high development cost for Adapter development and maintenance. Fortunately, the enterprise world has moved to a more *standards-based* approach and several patterns and best practices have evolved over the years. It has been shown that applying these standards and frameworks to the problem of tool integration is worthwhile and leads to a flexible and solid framework design for dynamic and open tool integration.

The proposed solution is highly portable and adaptable, meaning that any JBI-compliant solution can be used as a runtime and all Services and Adapters can be reused without change. The design minimizes cost in Adapter development and maximizes reuse of components and sharing of data and functionality. JNA has been successfully used for realizing the low-level protocol bridge to an existing COTS tool, Telelogic DOORS, without requiring complex and error-prone low-level integration code in the Adapter, despite being bound to the C library interface provided by DOORS. Also, existing scripts used to extend the DOORS interface and to connect to the integration framework (originally, ToolNet) could be reused without modification, demonstrating the flexibility and abstraction potential in the new, standards-based architecture.

The prototype's JMX-interface, originally used as a proxy for the ToolNet Desktop and seen more as a compromise, proved to be a viable replacement that could be extended to a full management console for administration and user control, providing rich access to framework services and configuration, including the installation, upgrade or removal of Adapters. Finally, Apache ServiceMix has shown to be a solid JBI implementation that provides a highly dynamic foundation for tool integration, with support for event-driven as well as service-oriented integration concepts and advanced, distributed and reliable messaging.

For real-world use of the proposed solution however, tool support still needs to improve, esp. during design time and debugging, in order to simplify Adapter development. This needs to happen in a standards-based way, so that development is not limited to particular JBI implementations with proprietary API extensions or special tooling. For this, the JBI standard itself needs to be revised, so as to redesign the API for easier component development, which is currently underway as part of *JSR-312* [JBI20].

The example of ToolNet has shown that realizing tool integration with a proprietary architecture built around the Eclipse framework introduces several limits in adaptability and dynamic, resulting in a more tightly coupled solution that is not easily portable, even among different Eclipse versions (see also Section 5.5). There is only one implementation of Eclipse (also the RCP framework used in the ToolNet desktop is inherently bound to the Eclipse project), which limits choice and makes the solution subject to design decisions and limitations of a single implementation and target group, without any alternatives. Also, despite its undisputed reputation as a tool integration platform, Eclipse follows the meta-tool approach, integrating tools into a central work environment, requiring refactoring of existing tools or relying on platform-specific facilities like OLE. This is not desired for a more transparent tool integration platform that keeps existing tools largely unchanged and also works with legacy COTS tools. Lastly, Eclipse does not provide an infrastructure for solving tool integration problems in this way, such as a common messaging backbone or extended Service querying, or a common concept for wrapping external systems beyond web services.

As a result of the research performed in the course of the prototype design and implementation, much of the custom design and proprietary solutions in current tool integration approaches can be replaced with standard

technologies and service-oriented integration concepts, as provided by JBI, building on best practices and existing standards in enterprise integration. These include WSDL-based component interfaces, loosely-coupled services and XML-based normalized messaging with rich routing and translation capabilities, which are also core concepts of an enterprise service bus. Consequently, many JBI implementations are based on an ESB-like message bus, but the JBI specification is open enough to allow for alternative, advanced topologies in a distributed environment, like *Grids* or *peer-to-peer* communication, and offers a transparent combination of *service-oriented* and *event-driven* architectures. Corresponding components and solutions are already available and provide integration designers and developers with a rich choice based on individual project requirements.

With all advances in integration standards and frameworks, one challenge in tool integration will always remain, as long as tools are not fully service-oriented and modularized themselves, as found by [Goose2000], who had similar problems integrating closed tools in the Microcosm-framework, in that “it is not possible to integrate closed tools in a general way. Each tool must be carefully analyzed and, depending on its nature, integrated in its own special way.”. This dilemma can only be solved in an effective and sustainable way by fully embracing open integration standards, patterns and frameworks that offer a flexible and lightweight Adapter architecture for integrating tools as-is, decoupling proprietary tool interfaces from the remaining, standards-based and open solution.

Appendix A. Prototype Source Excerpts

A.1. JBI Configuration

A.1.1. ToolNetServiceAssembly Descriptor

Example A.1: ServiceAssembly deployment descriptor jbi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi"
    xmlns:su1="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsServiceEngine"
    xmlns:su2="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsAdapter_Out"
    xmlns:su3="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsAdapter_In"
    xmlns:su4="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsServiceEngine_In"
    xmlns:su5="http://bostechcorp.com/SU/ToolNetServiceAssembly_Installer"
    version="1.0">
 <service-assembly>
  <identification>
   <name>ToolNetServiceAssembly</name>
   <description>ToolNetServiceAssembly</description>
  </identification>
  <service-unit>
   <identification>
    <name>ToolNetServiceAssembly_DoorsServiceEngine</name>
    <description>provides common ToolNet-services for DOORS</description>
   </identification>
   <target>
    <artifacts-zip>ToolNetServiceAssembly_DoorsServiceEngine.zip</artifacts-zip>
    <component-name>ToolNet-SE-Doors</component-name>
   </target>
  </service-unit>
  <service-unit>
   <identification>
    <name>ToolNetServiceAssembly_DoorsAdapter_Out</name>
    <description>realizes a socket-connection to Telelogic DOORS</description>
   </identification>
   <target>
    <artifacts-zip>ToolNetServiceAssembly_DoorsAdapter_Out.zip</artifacts-zip>
    <component-name>ToolNet-BC-DOORS</component-name>
   </target>
  </service-unit>
  <service-unit>
   <identification>
    <name>ToolNetServiceAssembly_DoorsAdapter_In</name>
    <description>realizes a socket-connection to Telelogic DOORS</description>
   </identification>
   <target>
    <artifacts-zip>ToolNetServiceAssembly_DoorsAdapter_In.zip</artifacts-zip>
    <component-name>ToolNet-BC-DOORS</component-name>
   </target>
  </service-unit>
  <service-unit>
   <identification>
    <name>ToolNetServiceAssembly_DoorsServiceEngine_In</name>
```

```
      <description>provides common ToolNet-services for DOORS</description>
    </identification>
    <target>
      <artifacts-zip>ToolNetServiceAssembly_DoorsServiceEngine_In.zip</artifacts-zip>
      <component-name>ToolNet-SE-Doors</component-name>
    </target>
  </service-unit>
  <service-unit>
    <identification>
      <name>ToolNetServiceAssembly_Installer</name>
      <description />
    </identification>
    <target>
      <artifacts-zip>ToolNetServiceAssembly_Installer.zip</artifacts-zip>
      <component-name>ChainBuilderESB-SE-Installer</component-name>
    </target>
  </service-unit>
  <connections>
    <connection>
      <consumer service-name="su1:ToolNetServiceAssembly_DoorsServiceEngine_Service"
            endpoint-name="ToolNetServiceAssembly_DoorsServiceEngine_Consumer" />
      <provider service-name="su2:ToolNetServiceAssembly_DoorsAdapter_Out_Service"
            endpoint-name="ToolNetServiceAssembly_DoorsAdapter_Out_Provider" />
    </connection>
    <connection>
      <consumer service-name="su3:ToolNetServiceAssembly_DoorsAdapter_In_Service"
            endpoint-name="ToolNetServiceAssembly_DoorsAdapter_In_Consumer" />
      <provider service-name="su4:ToolNetServiceAssembly_DoorsServiceEngine_In_Service"
            endpoint-name="ToolNetServiceAssembly_DoorsServiceEngine_In_Provider" />
    </connection>
  </connections>
 </service-assembly>
</jbi>
```

A.1.2. DoorsBindingComponent Descriptor

Example A.2: DoorsBindingComponent deployment descriptor jbi.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
 <component type="binding-component">
   <identification>
     <name>ToolNet-BC-DOORS</name>
     <description>realizes a socket-connection to Telelogic DOORS</description>
   </identification>

   <component-class-name>com.bostechcorp.cbesb.runtime.ccsl.base.CcslComponent</component-class-name>
   <component-class-path>
     <path-element>com.bostechcorp.cbesb.runtime.ccsl-base.jar</path-element>
     <path-element>DoorsBindingComponent.jar</path-element>
     <path-element>jna.jar</path-element>
     <path-element>com.bostechcorp.cbesb.runtime.component.util.jar</path-element>
   </component-class-path>

   <bootstrap-class-name>at.ac.tuwien.toolnet.adapter.doors.DoorsBootstrap</bootstrap-class-name>
   <bootstrap-class-path>
     <path-element>DoorsBindingComponent.jar</path-element>
     <path-element>com.bostechcorp.cbesb.runtime.ccsl-base.jar</path-element>
   </bootstrap-class-path>
```

```
  <shared-library>CCSL</shared-library>
 </component>
</jbi>
```

A.1.3. DoorsServiceEngine Descriptor

Example A.3: DoorsServiceEngine deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
 <component type="service-engine">
  <identification>
   <name>ToolNet-SE-Doors</name>
   <description>provides common ToolNet-services for DOORS</description>
  </identification>

  <component-class-name>com.bostechcorp.cbesb.runtime.ccsl.base.CcslComponent</component-class-name>
  <component-class-path>
   <path-element>com.bostechcorp.cbesb.runtime.ccsl-base.jar</path-element>
   <path-element>DoorsServiceEngine.jar</path-element>
   <path-element>com.bostechcorp.cbesb.runtime.component.util.jar</path-element>
  </component-class-path>

  <bootstrap-class-name>at.ac.tuwien.toolnet.adapter.doors.DoorsServiceEngineBootstrap</bootstrap-class-name>
  <bootstrap-class-path>
   <path-element>DoorsServiceEngine.jar</path-element>
   <path-element>com.bostechcorp.cbesb.runtime.ccsl-base.jar</path-element>
  </bootstrap-class-path>

  <shared-library>CCSL</shared-library>
 </component>
</jbi>
```

A.1.4. DoorsBindingComponent WSDL

Example A.4: Provider WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:de="http://cbesb.bostechcorp.com/dataenvelope/1.0"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsAdapter_Out"
          xmlns:DoorsAdapter="http://www.tuwien.ac.at/doors/1.0"
          name="ToolNetServiceAssembly_DoorsAdapter_Out"
          targetNamespace="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsAdapter_Out">
 <types>
  <xsd:schema xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd"
            targetNamespace="http://cbesb.bostechcorp.com/dataenvelope/1.0">
   <xsd:import namespace="http://ws-i.org/profiles/basic/1.1/xsd" />
   <xsd:element name="DataEnvelope">
    <xsd:complexType>
     <xsd:choice maxOccurs="unbounded">
      <xsd:element name="XMLRecord" type="xsd:anyType" />
      <xsd:element name="StringRecord" type="xsd:swaRef" />
      <xsd:element name="BinaryRecord" type="xsd:swaRef" />
     </xsd:choice>
    </xsd:complexType>
   </xsd:element>
```

```
  </xsd:schema>
 </types>
 <message name="DataEnvelopeMessage">
  <part name="body" element="de:DataEnvelope" />
 </message>
 <portType name="ToolNetServiceAssembly_DoorsAdapter_Out_Interface">
  <operation name="ToolNetServiceAssembly_DoorsAdapter_Out_Operation">
   <input message="tns:DataEnvelopeMessage" />
  </operation>
 </portType>
 <binding name="ToolNetServiceAssembly_DoorsAdapter_Out"
       type="tns:ToolNetServiceAssembly_DoorsAdapter_Out_Interface">
  <DoorsAdapter:binding />
 </binding>
 <service name="ToolNetServiceAssembly_DoorsAdapter_Out_Service">
  <port name="ToolNetServiceAssembly_DoorsAdapter_Out_Provider"
      binding="tns:ToolNetServiceAssembly_DoorsAdapter_Out">
   <DoorsAdapter:provider role="provider" DOORS_sender_port="5093" />
  </port>
 </service>
</definitions>
```

A.1.5. DoorsServiceEngine WSDL

Example A.5: Consumer WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
          xmlns:de="http://cbesb.bostechcorp.com/dataenvelope/1.0"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:tns="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsServiceEngine"
          xmlns:DoorsServiceEngine="http://www.tuwien.ac.at/toolnet/doorsserviceengine/1.0"
          name="ToolNetServiceAssembly_DoorsServiceEngine"
          targetNamespace="http://bostechcorp.com/SU/ToolNetServiceAssembly_DoorsServiceEngine">
 <types>
  <xsd:schema xmlns:ref="http://ws-i.org/profiles/basic/1.1/xsd"
           targetNamespace="http://cbesb.bostechcorp.com/dataenvelope/1.0">
   <xsd:import namespace="http://ws-i.org/profiles/basic/1.1/xsd" />
   <xsd:element name="DataEnvelope">
    <xsd:complexType>
     <xsd:choice maxOccurs="unbounded">
      <xsd:element name="XMLRecord" type="xsd:anyType" />
      <xsd:element name="StringRecord" type="xsd:swaRef" />
      <xsd:element name="BinaryRecord" type="xsd:swaRef" />
     </xsd:choice>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>
 </types>
 <message name="DataEnvelopeMessage">
  <part name="body" element="de:DataEnvelope" />
 </message>
 <portType name="ToolNetServiceAssembly_DoorsServiceEngine_Interface">
  <operation name="ToolNetServiceAssembly_DoorsServiceEngine_Operation">
   <input message="tns:DataEnvelopeMessage" />
   <output message="tns:DataEnvelopeMessage" />
  </operation>
 </portType>
 <binding name="ToolNetServiceAssembly_DoorsServiceEngine"
```

```
        type="tns:ToolNetServiceAssembly_DoorsServiceEngine_Interface">
   <DoorsServiceEngine:binding />
 </binding>
 <service name="ToolNetServiceAssembly_DoorsServiceEngine_Service">
   <port name="ToolNetServiceAssembly_DoorsServiceEngine_Consumer"
         binding="tns:ToolNetServiceAssembly_DoorsServiceEngine">
     <DoorsServiceEngine:consumer role="consumer" triggerInterval="5000"
                           dxlCommandSend="ack &amp;quot;Hello DOORS from JBI!&amp;quot;"
                           defaultMep="in-only" />
   </port>
 </service>
</definitions>
```

A.2. JBI Adapter Implementation

This section shows relevant parts of the custom JBI component implementation, comprising the DoorsBinding-Component and the DoorsServiceEngine. Logging and error handling has been stripped to make the code more readable and easier to follow.

A.2.1. DoorsBindingComponent

A.2.1.1. JNA Interface used in the DoorsBindingComponent

Example A.6: JNA interface wrapper for the DOORS API

```
package at.ac.tuwien.toolnet.adapter.doors;

import com.sun.jna.*;

public interface DoorsLibrary extends Library {
	public final static int DOORS_API_OK = 0;
	public final static int DOORS_API_PARSE_BAD_DXL = 1;
	public final static int DOORS_API_SEND_BAD_DXL = 2;
	public final static int DOORS_API_CONNECT_FAILED = 3;
	public final static int DOORS_API_ERROR = 4;

	/**
	 * sends error message to DOORS
	 *
	 * @param format
	 *            error message
	 */
	public void apiError(String format);

	/**
	 * initializes DOORS C-API
	 *
	 * @param n
	 *            name of the resulting language, Null for DXL
	 * @param ext
	 *            name of scripts filename extension, Null for .cdi
	 * @param include
	 *            searchpath for include files, Null value defaults current
	 *            directory
	 */
	public void apiInitLibrary(final String n, final String ext,
			final String include);
```

```
/**
 * Sets whether the API functions produce error messages on the command
 * line. By default, the functions produce command line errror messages, but
 * you can prevent that using this function
 *
 * @param onOFF
 */
public void apiQuietError(final int onOFF);

/**
 * Sets whether the API functions exit on error. by default, the functions
 * exit, but you can prevent that using this function
 *
 * @param onOFF
 */
public void apiExitOnError(final int onOFF);

/**
 * opens TCP/IP Socket connection to DXL-server
 *
 * @param portNum
 *            port number
 * @param hostAdress
 *            host name of remote machine
 */
public void apiConnectSock(final int portNum, String hostAdress);

/**
 * sends string to open socket connection, which is interpreted as script by
 * DOORS
 *
 * @param format
 *            the script to be interpreted by doors
 */
public void apiSend(final String format);

/**
 * sends string to open socket connection, which is interpreted as script by
 * DOORS
 *
 * @param tmt
 *            timeout in seconds
 * @param format
 *            the script to be interpreted by doors
 */
public void apiSendTimesout(final int tmt, final String format);

/**
 * winds down the C-API
 */
public void apiFinishLibrary();

/**
 * returns the error that occured most recently
 *
 * @see DoorsCAPIWrapper#DOORS_API_OK
 * @see DoorsCAPIWrapper#DOORS_API_PARSE_BAD_DXL
 * @see DoorsCAPIWrapper#DOORS_API_SEND_BAD_DXL
 * @see DoorsCAPIWrapper#DOORS_API_CONNECT_FAILED
```

```
        * @see DoorsCAPIWrapper#DOORS_API_ERROR
        */
       public int apiGetErrorState();
}
```

A.2.1.2. DoorsEndpoint

Example A.7: DoorsEndpoint implementation realizing the JMX connection

```
package at.ac.tuwien.toolnet.adapter.doors;

[import related DoorsAdapter Java classes]
import com.bostechcorp.cbesb.runtime.ccsl.jbi.messaging.*;
[import javax...]

public class DoorsEndpoint extends ScheduledEndpointProcessor {
/**
 * JNA interface to the DOORS API library
 */
protected ComponentContext context;
protected String doorsPort = "";
private DoorsConsumerListener doorsListenerThread = null;
/**
 * reference to the JMX MBean of this BindingComponent,
 * can be either a ProviderMBean or a ConsumerMBean
 */
private DoorsConfiguration configurationMBean = null;
/**
 * the JMX ObjectName of the Configuration MBean
 */
private ObjectName configurationMBeanName = null;
/**
 * MBean allowing configuration of the Doors BC-Provider
 *
 * allows to set the Listener and Sender port when connecting
 * to the Telelogic DOORS application
 */
private static final int MBEAN_PROVIDER = 0;
/**
 * sets up the endpoint handler and JMX MBean
 */
public DoorsEndpoint() {
   super();
}
/* Setter for the DOORS_receiver_port */
public void setDoorsPort(String value) {
   this.doorsPort = value;
}
/* Getter for the DOORS_receiver_port*/
public String getDoorsPort() {
   return this.doorsPort;
}

protected IComponentProcessor createProviderProcessor() {
   DoorsProviderProcessor provider = new DoorsProviderProcessor(this);
   provider.setMessageExchangeFactory(exchangeFactory);
   provider.setChannel(channel);
   provider.setContext(context);
   return provider;
```

```
}

protected IComponentProcessor createConsumerProcessor() {
   setHandler(new DoorsConsumerHandler(this));
   return new CbEmbeddedSchedulerConsumerProcessor(this);
}

public void setContext(ComponentContext context) {
   this.context = context;
}

@Override
public void start() throws Exception {
   // perform Role-specific configuration and setup
   if (getRole() == MessageExchange.Role.CONSUMER) {
      // if Consumer, start DOORS listener Thread
      doorsListenerThread = new DoorsConsumerListener(this);
      doorsListenerThread.start();
   } else {
      // register this endpoint for management access
      registerConfigurationMBean(MBEAN_PROVIDER);
   }
}

@Override
public void stop() throws Exception {
   // unregister JMX MBean
   unregisterConfigurationMBean();
   if (doorsListenerThread != null) {
      doorsListenerThread.shutdown();
   }
}

public DoorsConsumerListener getDoorsListenerThread() {
   return doorsListenerThread;
}
// JMX management methods
/**
 * sets up a new MBean for configuring the DOORS component
 * @return
 */
private ObjectName createConfigurationMBeanName(String name) {
   logger.debug("creating ConfigurationMBean " + name + " in context " + this.context);
   return this.context.getMBeanNames().createCustomComponentMBeanName(name);
}
/**
 * make available configuration and control
 * of the DOORS Adapter for JMX management access
 * @throws JBIException
 * @see #start()
 */
private void registerConfigurationMBean(int type) throws JBIException {
   // set up MBean if necessary
   if (configurationMBeanName == null) {
      if (type == MBEAN_PROVIDER) {
         configurationMBeanName = createConfigurationMBeanName("DOORS-Sender");
         configurationMBean = new DoorsConfiguration(this);
      } else {
         configurationMBeanName = createConfigurationMBeanName("DOORS-Receiver");
         configurationMBean = new DoorsConfiguration(this);
```

```
        }
    }
    StandardMBean mbean = new StandardMBean(configurationMBean, DoorsConfigurationMBean.class);
    server.registerMBean(mbean, this.configurationMBeanName);
}
/**
 * remove MBean from management access
 * @see #stop()
 */
private void unregisterConfigurationMBean() {
    // unregister Configuration-MBean
    MBeanServer server = this.context.getMBeanServer();
    if (server.isRegistered(this.configurationMBeanName)) {
        server.unregisterMBean(this.configurationMBeanName);
    }
}
```

A.2.1.3. DoorsBindingComponent (Consumer)

Example A.8: DoorsConsumerListener routing incoming calls to the JBI message router

```
/**
* The Consumer part listens for incoming commands from the Tool-Side DOORS Adapter
*/
package at.ac.tuwien.toolnet.adapter.doors.processors;

[imports...]
import at.ac.tuwien.toolnet.adapter.doors.DoorsEndpoint;
// Chainbuilder extensions
import com.bostechcorp.cbesb.runtime.ccsl.lib.ExternalInput;
import com.bostechcorp.cbesb.runtime.component.util.wsdl.WsdlMepConstants;
/**
* waits for incoming DOORS connections
* and translates calls into normalized messages, creating a new NormalizedMessageExchange
*/
public class DoorsConsumerListener extends Thread {
    private static final String SENDER_ENDPOINT_PROPERTY = "org.apache.servicemix.senderEndpoint";
    private boolean isRunning = true;
    private DoorsEndpoint endpoint;
    ServerSocket doorsSocket = null;
    Socket s = null;
    int doorsPort = 0;

    public DoorsConsumerListener(DoorsEndpoint endpoint) {
        this.endpoint = endpoint;
    }
    /*
    * Thread main method
    */
    public void run() {
        StringBuffer dxlbuffer = null;
        String line;
        // now setup socket and wait for "response"
        doorsSocket = new ServerSocket(doorsPort);
        while (isRunning) {
            // wait for new connection from DOORS
            s = doorsSocket.accept();

            InputStreamReader is = null;
```

```
        BufferedReader reader = null;
        // we received input from DOORS
        is = new InputStreamReader(s.getInputStream());
        reader = new BufferedReader(is);
        dxlbuffer = new StringBuffer();
        while (!s.isClosed() && (line = reader.readLine()) != null) {
           dxlbuffer.append(line);
        }
        s.close();
        // create inbound message exchange from input received
        createInbound(dxlbuffer.toString().getBytes());
    } // while
    // shut down
    doorsSocket.close();
    doorsSocket = null;
}
public void shutdown() {
    isRunning = false;
}
public void forceStop() {
    isRunning = false;
    doorsSocket.close();
}
/**
* create inbound message exchange for external input from socket
*/
public byte[] createInbound(byte[] bytes) throws MessagingException, Exception {
    byte[] returnBytes = null;
    MessageExchange me = null;
    DeliveryChannel channel = endpoint.getChannel();
    ComponentContext context = endpoint.getServiceUnit().getComponent().getComponentContext();

    // create a message exchange
    URI defaultMep = endpoint.getDefaultMep();
    if (defaultMep.compareTo(WsdlMepConstants.IN_ONLY) == 0) {
        me = channel.createExchangeFactory().createInOnlyExchange();
    } else if (defaultMep.compareTo(WsdlMepConstants.IN_OUT) == 0) {
        me = channel.createExchangeFactory().createInOutExchange();
    } else if (defaultMep.compareTo(WsdlMepConstants.ROBUST_IN_ONLY) == 0) {
        me = channel.createExchangeFactory().createRobustInOnlyExchange();
    } else
        throw new Exception("trying to process unknown MEP \""+defaultMep+"\"");

    // populate the exchange and send it into the container
    me.setOperation(endpoint.getDefaultOperation()); // there is no getOperationQName();
    String endpointKey = "{" + endpoint.getService().getNamespaceURI() + "}"+
        endpoint.getService().getLocalPart() + ":" + endpoint.getEndpoint();
    me.setProperty(SENDER_ENDPOINT_PROPERTY, endpointKey);
    ExternalInput ext = new ExternalInput(new ByteArrayInputStream(bytes),
            "UTF-8", "raw", "string", 0);
    // now create a new JBI NormalizedMessage and send it to the bus
    NormalizedMessage msg = me.createMessage();
    ext.populateMessage(msg);
    // create a new IN exchange
    me.setMessage(msg, "in");
    // set the target endpoint (could be queried dynamically)
    ServiceEndpoint linkedEndpoint = context.getEndpoint(endpoint.getService(), endpoint.getEndpoint());
    me.setEndpoint(linkedEndpoint);
    me.setService(endpoint.getService());
```

```
      // do an asynchronous send, no return bytes
      channel.send(me);
      return returnBytes;
   }
}
```

A.2.1.4. DoorsBindingComponent (Provider)

Example A.9: The DoorsProviderProcessor routes JBI messages to DOORS

```
package at.ac.tuwien.toolnet.adapter.doors.processors;
[imports...]
import at.ac.tuwien.toolnet.adapter.doors.DoorsEndpoint;
import at.ac.tuwien.toolnet.adapter.doors.DoorsLibrary;
import at.ac.tuwien.toolnet.adapter.doors.DoorsLibraryFactory;

import com.bostechcorp.cbesb.runtime.ccsl.jbi.messaging.CbProviderProcessor;
import com.bostechcorp.cbesb.runtime.ccsl.lib.DumpNormalizedMessage;
import com.bostechcorp.cbesb.runtime.ccsl.nmhandler.NormalizedMessageHandler;
import com.bostechcorp.cbesb.runtime.ccsl.nmhandler.StringSource;

public class DoorsProviderProcessor extends CbProviderProcessor {
   private DeliveryChannel channel;
   private MessageExchangeFactory messageExchangeFactory;
   protected ComponentContext context;
   DoorsEndpoint endpoint;
   /**
   * default client port for client-connections to DOORS
   */
   int doorsPort = 5093;
   [component setup methods...]
   /**
   * transmits a DXL-script contained in a JBI message to DOORS
   * using the DOORS C API for communicating over a TCP/IP socket
   */
   @Override
   public void processInMessage(QName service, QName operation,
         NormalizedMessage in, MessageExchange exchange) throws Exception {
      // get the JNA library stub for the DOORS lib
      DoorsLibrary lib = DoorsLibraryFactory.getInstance();
      // open IPC connection to DOORS @todo make host configurable, too
      lib.apiConnectSock(doorsPort, "127.0.0.1");
      int stat = lib.apiGetErrorState();
      if (stat == DoorsLibrary.DOORS_API_OK) {
         // get message content string using ChainBuilder's utility class
         NormalizedMessageHandler nmh = new NormalizedMessageHandler(in);
         Source src = nmh.getRecordAtIndex(0);
         if (src instanceof StringSource) {
            StringSource strsrc = (StringSource) src;
            String dxl = strsrc.getText();
            // send in message to DOORS
            lib.apiSendTimesout(300, dxl);
            // close connection again
            lib.apiSendTimesout(100, "quit_");
            // check return status
            stat = lib.apiGetErrorState();
            if (stat == DoorsLibrary.DOORS_API_OK) {
               /*
               * exchange.setStatus(ExchangeStatus.DONE);
```

```
                */
            } else {
                // something went wrong
            }
        } else {
            // unknown format
        }
    } else {
        // process any errrs that may haved occured
    }
  }
  // [processing other MessageExchangePatterns like IN_OUT stripped]
}
```

A.2.1.5. BindingComponentMBean Definition

This MBean allows configuration of the DoorsBindingComponent, e.g. DOORS server and port.

Example A.10: DoorsConfigurationMBean for configuring the DoorsBindingComponent

```
package at.ac.tuwien.toolnet.adapter.doors;

public interface DoorsConfigurationMBean {
  /**
  * start DOORS from the management console
  */
  public void startDoors();
  /**
  * stop the running DOORS instance
  */
  public void stopDoors();
  /**
  * sends a message to a running DOORS instance
  * at the port configured
  * @return <code>true</code> if the message was sent successfully
  */
  public String sendMessage(String message);
  public int getClientPort();
  public void setClientPort(int clientPort);
  public int getServerPort();
  public void setServerPort(int serverPort);
}
```

A.2.2. DoorsServiceEngine

A.2.2.1. DoorsServiceEngine (Consumer)

The class `DoorsServiceEngineConsumerListener.java` implements a simple translator that receives Service requests from the JMX console that acts as a ToolNetDesktop-replacement and forwards it to the Doors-BindingComponent.

Example A.11: DoorsServiceEngine Consumer implementation

```
package at.ac.tuwien.toolnet.adapter.doors.processors;
[import ...]
/**
 * handles commands received from JMX, i.e. user input from the ToolNet/JBI "console"
 */
```

```
public class DoorsServiceEngineConsumerHandler extends ScheduledProcessHandler {
	DoorsServiceEngineEndpoint endpoint;
	/**
	* constant for creating the MessageExchange in {@link #sendMessage(String)}
	*/
	private static final String SENDER_ENDPOINT_PROPERTY = "org.apache.servicemix.senderEndpoint";
	private static final String DOORS_ENDPOINT_BASE = "ToolNetServiceAssembly_DoorsAdapter_Out";
	private static final String DOORS_ENDPOINT_NAME = DOORS_ENDPOINT_BASE + "_Provider";
	private static final String DOORS_SERVICE_NAME = DOORS_ENDPOINT_BASE + "_Service";
	private static final String DOORS_SERVICE_URL = "http://bostechcorp.com/SU/" + DOORS_ENDPOINT_BASE;

	public DoorsServiceEngineConsumerHandler(DoorsServiceEngineEndpoint endpoint) {
		super(endpoint);
		this.endpoint = endpoint;
	}
	// here we translate from the common ToolNet Service to the tool-specific action, using DOORS DXL
	public void highlightObject(String module, int no) {
		String dxl = "#include <addins/ToolNet/ToolNet_PresentationService.inc>;" +
				"ToolNet_IPresentation_showObject(\"" + module + "\",\"" + no + "\"," +
				"\"null\",\"null\",\"HIGHLIGHT_OBJECT\")";
		sendMessage(dxl);
	}
	/**
	 * send message to DOORS BC
	 */
	public void sendMessage(String message) {
		MessageExchange me = null;
		NormalizedMessage msg = null;
		DeliveryChannel channel = endpoint.getChannel();
		ComponentContext context = endpoint.getServiceUnit().getComponent().getComponentContext();
		// target endpoint (=DOORS BC Out)
		ServiceEndpoint linkedEndpoint = context.getEndpoint(
			new QName(DOORS_SERVICE_URL, DOORS_SERVICE_NAME),
			DOORS_ENDPOINT_NAME);
		try {
			// create a message exchange (only IN-ONLY supported)
			me = channel.createExchangeFactory().createInOnlyExchange();
			msg = me.createMessage();
			me.setOperation(endpoint.getDefaultOperation());
			String endpointKey = "{" + endpoint.getService().getNamespaceURI() + "}"+
			   endpoint.getService().getLocalPart() + ":" + endpoint.getEndpoint();
			me.setProperty(SENDER_ENDPOINT_PROPERTY, endpointKey);
			// the CBESB-Helperclass NormalizedMessageHandler wraps the NormalizedMessage
			NormalizedMessageHandler msghandler = new NormalizedMessageHandler(msg);
			// Add the Source as a record
			msghandler.addRecord(new StringSource(message));
			msg = msghandler.generateMessageContent();
			me.setMessage(msg, "in");
			me.setEndpoint(linkedEndpoint);
			me.setService(endpoint.getService());
			/*
			 * do asynchronous send
			 * (better would be synchronous send to check reply from DOORS-BC,
			 *  but in non-batch-mode, DOORS blocks on a dialog boxes and
			 *  thus produces a timeout during MessageExchange, anyway)
			 */
			channel.send(me);
		} catch (MessagingException e) {
			logger.fatal("Could not send message due to a messaging error: ", e);
		} catch (Exception e) {
```

```
                logger.fatal("Could not send message due to error: ", e);
            }
        }
}
```

A.2.2.2. DoorsServiceEngine (Provider)

Example A.12 below shows the relevant parts of the prototype DoorsAdapter ServiceEngine, automatically generated getter and setter methods were ommitted, as well as parsing the input String received from DOORS, and error handling was minimized for the sake of clarity.

Example A.12: ServiceEngine implementation DoorsServiceEngineProviderProcessor.java

```
package at.ac.tuwien.toolnet.adapter.doors.processors;
[...]
public class DoorsServiceEngineProviderProcessor extends CbProviderProcessor {
    private DeliveryChannel channel;
    private MessageExchangeFactory messageExchangeFactory;
    protected ComponentContext context;
    DoorsServiceEngineEndpoint endpoint;
    /**
    * holds references to managed DOORS Object-MBeans
    * for later lookup on showObject-requests from the DOORS Adapter
    */
    HashMap<Integer, ObjectName> doorsMBeanNames = new HashMap<Integer, ObjectName>();
    public static final String TOOLNET_TOOL_STARTED =
        "org.toolnet.core.model.other.ILocalToolNet:toolStarted()";
    public static final String ToolNet_PresentationService_SHOWOBJECT =
        "org.toolnet.core.model.services.IPresentation:showObject";
    public static final String ToolNet_RelationCreationInterface =
        "org.toolnet.core.model.services.IRelationCreation";
    public static final String ToolNet_RelationCreationClient_ADDANCHOR = "addAnchor";
    public static final String ToolNet_RelationCreationClient_REMOVEANCHORS = "removeAnchors";

    public DoorsServiceEngineProviderProcessor(
            DoorsServiceEngineEndpoint endpoint) {
        super(endpoint);
        this.endpoint = endpoint;
    }
    @Override
    /**
    * processes an incoming DOORS-call and translates it the
    * corresponding ToolNet Service-invocation
    */
    public void processInMessage(QName service, QName operation,
            NormalizedMessage in, MessageExchange exchange) throws Exception {
        // get message content string (standard JBI way: DOM-Transformer-variant from ServiceMix-project)
        NormalizedMessageHandler nmh = new NormalizedMessageHandler(in);
        Source src = nmh.getRecordAtIndex(0);
        if (! (src instanceof StringSource)) {
            // unexpected format
            return;
        }
        // extract ToolNet service
        StringSource strsrc = (StringSource) src;
        String request = strsrc.getText();
        // parse input command and react on it
        if (request.equalsIgnoreCase(TOOLNET_TOOL_STARTED)) {
            logger.debug("DOORS Adapter started successfully");
```

```
        } else if (request.startsWith(ToolNet_PresentationService_SHOWOBJECT)) {
            logger.debug("Show Object requested - not implemented yet.");
        } else if (request.contains(ToolNet_RelationCreationClient_ADDANCHOR)) {
            logger.debug("DOORS Adapter addAnchor requested:");
            // parse request from DOORS Adapter, looks like:
            // addAnchor((id)["00000661","34","__NULL__","__NULL__"],(AddAsType)"1")
            String module;
            // parse module = 1st parameter inside brackets
            ...
            // scan from second parameter inside brackets: ["first","second"...
            ...
            // parse Type
            ...
            // register MBean for accessing the DOORS Object from a JMX-console
            registerDoorsMBean(module, id, type);
        } else if (request.contains(ToolNet_RelationCreationClient_REMOVEANCHORS)) {
            logger.debug("DOORS Adapter addAnchor requested:");
            // parse ID
            // register MBean for accessing the DOORS Object from a JMX-console
            unregisterDoorsMBean(id);
        } else if (request.startsWith("return")) {
            // parse return operation
            logger.debug("DOORS-Adapter successfully invoked operation: " + op);
        } else {
            logger.warn("Unknown command received and ignored.");
        }
        // done with ME
        logger.debug("DONE with MessageExchange");
        /*
        * JBI spec requires manually setting status, but
        * Chainbuilder-lib does this automatically:
        * exchange.setStatus(ExchangeStatus.DONE);
        */
    }
    ...
    /**
    * make available configuration and control
    * of a DOORS Object for JMX management access
    *
    * @throws JBIException
    * @see #start()
    */
    private void registerDoorsMBean(String module, int id, int type) throws JBIException {
        // set up MBean if necessary
        ObjectName mbn = this.context.getMBeanNames().createCustomComponentMBeanName("DOORS Object #"+id);
        // first register in internal registry
        doorsMBeanNames.put(new Integer(id), mbn);
        // then register in MBeanServer
        StandardMBean mbean = new StandardMBean(new DoorsObject(
                        (DoorsServiceEngineEndpoint) getEndpoint(), module, id, type),
                        DoorsObjectMBean.class);
        if (mbean != null) {
            MBeanServer server = this.context.getMBeanServer();
            server.registerMBean(mbean, mbn);
        }
    }
    /**
    * remove MBean from management access
    * @see #stop()
    */
```

```
    private void unregisterDoorsMBean(int id) {
        // get ObjectName for ID
        ObjectName mbn = doorsMBeanNames.get(id);
        // unregister Configuration-MBean
        MBeanServer server = this.context.getMBeanServer();
        if (server.isRegistered(mbn)) {
            server.unregisterMBean(mbn);
        }
    }
}
```

A.2.2.3. DoorsObjectMBean (ServiceEngine MBean)

This MBean represents a requirements object in DOORS and allows viewing and changing the object's properties. Also the highlight()-service can be invoked on the selected requirement object (MBean), as needed for the prototype scenario.

Example A.13: The DoorsObjectMBean interface

```
package at.ac.tuwien.toolnet.adapter.doors.ui;

public interface DoorsObjectMBean {
   public int    getId();
   public String getModule();
   public String getName();
   public void   setName(String name);
   public String getDescription();
   public int    getAnchorType();
   public String getAnchorTypeName();
   /**
    * brings this object into focus
    */
   public void highlight();
}
```

Example A.14: The DoorsObject implementation

```
package at.ac.tuwien.toolnet.adapter.doors.ui;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import at.ac.tuwien.toolnet.adapter.doors.DoorsServiceEngineEndpoint;

public class DoorsObject implements DoorsObjectMBean {
    public static final int ToolNet_RelationCreation_TARGET = 0;
    public static final int ToolNet_RelationCreation_SOURCE = 1;
    public static final int ToolNet_RelationCreation_IDENTITY = 2;
    protected final transient Log logger = LogFactory.getLog(getClass());
    DoorsServiceEngineEndpoint endpoint;
    private int id;
    private String module;
    private int anchorType;
    private String name;
    private String description;

    /**
    * connects to the endpoint to be managed by this MBean
    * @param endpoint
```

```
    */
    public DoorsObject(DoorsServiceEngineEndpoint endpoint,
                String module, int id, int type, String name, String description) {
        logger.debug("init ObjectMBean for ConsumerHandler (" +
                    "ID="+id+", type=" +type+")");
        assert(endpoint != null):"No connection to endpoint!";
        this.endpoint = endpoint;
        this.module = module;
        this.id = id;
        this.anchorType = type;
        this.name = name;
        this.description = description;
    }
    public DoorsObject(DoorsServiceEngineEndpoint endpoint, String module, int id, int type) {
        this(endpoint, module, id, type, "unknown", "empty");
    }
    public String getDescription() {
        return description;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public int getAnchorType() {
        return anchorType;
    }
    public String getAnchorTypeName() {
        if (this.anchorType == ToolNet_RelationCreation_TARGET)
            return "Target";
        else if (this.anchorType == ToolNet_RelationCreation_SOURCE)
            return "Source";
        else if (this.anchorType == ToolNet_RelationCreation_IDENTITY)
            return "Identity";
        else
            return "unknown";
    }
    public void highlight() {
        this.endpoint.getHandler().highlightObject(getModule(), getId());
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getModule() {
        return this.module;
    }
}
```

A.3. Existing Tool-Side DOORS Adapter

The following sections show existing scripting code as part of the Tool-side Adapter that is integrated into the DOORS interface.

A.3.1. ToolNet Menu Definition

For integrating the ToolNet-Adapter into the DOORS-interface, a custom menu is defined as shown in Example A.15 below, according to the specification of the menu definition format described in [DXL], chapter *DOORS*

window control: Each line (except for the separators) starts with a function name that implements the menu operation, then a shortcut can be defined, and lastly the menu item label is defined.

Example A.15: ToolNet menu definition from ToolNet.idx

```
ToolNet_startLink   s _ Set object as source
ToolNet_endLink    e _ Set object as target
-------------------------------------------------------------------
ToolNet_relation          R _ Get relations for object
ToolNet_markLinked  M _ Mark linked object
-------------------------------------------------------------------
ToolNet_showObject       o _ Goto object
ToolNet_highlightObject h _ Highlight object
-------------------------------------------------------------------
ToolNet_addObjectToGroup            A _ Add object to group
ToolNet_markGroupObjects            K _ Mark objects of the group
ToolNet_markGroupLinkedObjects J _ Mark Objects linked to the group
ToolNet_groupRelation                L _ Get relations for group objects
-------------------------------------------------------------------
ToolNet_callWindow            w _ ToolNet window
-------------------------------------------------------------------
ToolNet_DoorsRelationCreationClient1     Q _ Export Links
```

The prototype use case covers the creation of Relations, which are represented by the first two menu commands covered in Section A.3.3.

A.3.2. ToolNet IPC implementation

The TCP/IP-based inter-process communication between DOORS and the JBI prototype is implemented in Example A.16 below (helper methods omitted). Other ToolNet-scripts rely on the functions defined therein to send requests to the ToolNetSide DOORS-Adapter (realized by the DoorsBindingComponent in the prototype).

Example A.16: DXL source of ToolNet_ipc.inc

```
/**
* Script: IPC-communication for ToolNet-Framework
*
* Datum *    Änderungsbeschreibung                    * Autor *
* 31.10.01  Ersterstellung                        Jürgen Großmann
*/
// NEW TOOL <> TOOLADAPTER COMMU
// ipc functions
void ToolNet_ipc_send(string);
void ToolNet_ipc_sendRequest(string, string, string);
void ToolNet_ipc_sendReturn(string, string);
void ToolNet_ipc_sendVoidReturn(string);

/**
 * The given message is sent to the ToolAdapter.
 */
void ToolNet_ipc_send(string i_message) {
    ToolNet_ack("\nSENDING:\n" i_message "\n");
    IPC javaSocket;
    javaSocket = client(ToolNet_client_port,ToolNet_client_localhost);
    if(! null javaSocket) {
        send(javaSocket, i_message "\n");
        delete(javaSocket);
```

```
    } else {
        // communication error
    }
}

/**
 * Creates a request to be handled by the given method of the given interface.
 * The resulting message is sent to the ToolAdapter.
 */
void ToolNet_ipc_sendRequest(string i_interface, string i_method, string i_parameters) {
    string message = i_interface ":" i_method "(" i_parameters ")";
    ToolNet_ipc_send(message);
}

/**
 * Creates a parameterless request to be handled by the given method of the given interface.
 * The resulting message is sent to the ToolAdapter.
 */
void ToolNet_ipc_sendVoidRequest(string i_interface, string i_method) {
    ToolNet_ipc_sendRequest(i_interface,i_method,"");
}

/**
 * Creates a return message with the given message identifier (ACT).
 * The resulting message is sent to the ToolAdapter with parameters.
 */
void ToolNet_ipc_sendReturn(string i_identifier, string i_parameters) {
    string message = "return " i_identifier ": " i_parameters "";
    ToolNet_ipc_send(message);
}

/**
 * Creates a void return message with the given message identifier (ACT).
 */
void ToolNet_ipc_sendVoidReturn(string i_identifier) {
    string message = "return " i_identifier;
    ToolNet_ipc_send(message);
}
```

A.3.3. ToolNet RelationService implementation in DOORS

The menu operation ToolNet_startLink calls the DXL-function `addCurrentObjectAsAnchor()`-function to create a Link Source as shown below:

Example A.17: Implementation of ToolNet_startLink in ToolNet_startLink.dxl:

```
// sets ToolNet Link
/*
  This script sets the source of a ToolNet Link
*/
//# main
#include <addins/ToolNet/ToolNet_RelationCreationClient.inc>

ToolNet_RelationCreationClient_addCurrentObjectAsAnchor(ToolNet_RelationCreation_SOURCE);
```

Similarly, the menu operation ToolNet_endLink calls the `addCurrentObjectAsAnchor()`-function to define a Link Target:

Example A.18: Implementation of ToolNet_endLink in ToolNet_endLink.dxl:

```
// sets ToolNet Link
/*
  This script sets the target of  ToolNet Link
*/
//# main
#include <addins/ToolNet/ToolNet_RelationCreationClient.inc>

ToolNet_RelationCreationClient_addCurrentObjectAsAnchor(ToolNet_RelationCreation_TARGET);
```

The following source shows the implementation of the function `addCurrentObjectAsAnchor()` that eventually sends a ToolNet-request over the IPC-channel to the prototype:

Example A.19: Implementation of ToolNet_PresentationClient.inc

```
/**
* Script: Manage relation client for ToolNet-Framework,
* creates relation in ToolNet-Framework
*************************************************************************
* Datum *   Änderungsbeschreibung                        * Autor *
*************************************************************************
* 31.10.01  Ersterstellung                        Jürgen Großann
* 21.07.03  Implementing new Tool / Tool-Adapter communiation
*          Stephan Weiß
*************************************************************************/
const string ToolNet_RelationCreationInterface = "org.toolnet.core.model.services.IRelationCreation"

const string ToolNet_RelationCreationClient_ADDANCHOR = "addAnchor"
const string ToolNet_RelationCreationClient_REMOVEANCHORS = "removeAnchors"

const string ToolNet_RelationCreation_TARGET = "0"
const string ToolNet_RelationCreation_SOURCE = "1"
const string ToolNet_RelationCreation_IDENTITY = "2"

/*************************************************************************
* Funktionsname:       ToolNet_RelationCreationClient_addAnchor
* Zweck:               adds an anchor for a link
*************************************************************************
* Datum *   Änderungsbeschreibung                        * Autor *
*************************************************************************
* 23.09.03  Ersterstellung                        Stephan Weiss
*************************************************************************/
void ToolNet_RelationCreationClient_addAnchor(string refTokens[], string anchorType) {

   string parameters = ToolNet_arg_asTypedStringArray(refTokens,"id")
   parameters = parameters "," ToolNet_arg_asTypedString(anchorType,"AddAsType");
   ToolNet_ipc_sendRequest(  ToolNet_RelationCreationInterface,
                        ToolNet_RelationCreationClient_ADDANCHOR,
                        parameters );
}

/*************************************************************************
* Funktionsname:       ToolNet_RelationCreationClient_addAnchor
* Zweck:               adds an anchor for a link
*************************************************************************
* Datum *   Änderungsbeschreibung                        * Autor *
*************************************************************************
* 23.09.03  Ersterstellung                        Stephan Weiss
*************************************************************************/
```

```
void ToolNet_RelationCreationClient_addAnchor(Module mod, Object obj,
                                string anchorType) {
   string refTokens[4];
   ToolNet_idmap_writeObjectID(mod,obj,refTokens);
   ToolNet_RelationCreationClient_addAnchor(refTokens, anchorType);
}

/*****************************************************************************
* Funktionsname:       ToolNet_RelationCreationClient
*                      _addCurrentObjectAsAnchor
* Zweck:               adds an anchor for a link
*****************************************************************************
* Datum *   Änderungsbeschreibung                    * Autor *
*****************************************************************************
* 23.09.03  Ersterstellung                           Stephan Weiss
*****************************************************************************/
void ToolNet_RelationCreationClient_addCurrentObjectAsAnchor(string anchorType) {
   ToolNet_RelationCreationClient_addAnchor(current Module, current Object, anchorType);
}
```

A.3.4. ToolNet PresentationService implementation in DOORS

Example A.20 shows the implementation of the ToolNet SHOWOBJECT-Service provided by the DoorsAdapter, which highlights an Object in the DOORS interface:

Example A.20: Implementation of ToolNetPresentationService.inc

```
/*****************************************************************************
* Script: Presentation service for ToolNet-Framework,
*         receive presentation commands
*****************************************************************************
* Datum *   Änderungsbeschreibung                    * Autor *
*****************************************************************************
* 31.10.01  Ersterstellung                           Jürgen Großmann
*
*****************************************************************************/

/*****************************************************************************
* Funktionsname:       ToolNet_PresentationService_showObject
* Eingang:             string i_modID,
*                      string i_objID,
*                      string i_attribute,
*                      string i_offset
* Ausgang:             -
* Zweck:               gets request for SHOWOBJECT an shows
*                      specified object
*****************************************************************************
* Datum *   Änderungsbeschreibung                    * Autor *
*****************************************************************************
* 31.10.01  Ersterstellung                           Jürgen Großmann
* 07.01.03  changed for use with filters and views   Stephan Weiss
*****************************************************************************/
void ToolNet_IPresentation_showObject ( string i_modID, string i_objID,
                                string i_attribute, string i_offset,
                                string i_act ) {
 Object objToShow;
 Module mdlToWorkIn;
```

```
 ToolNet_ack ("showObject " i_modID ", " i_objID ", " i_attribute ", " i_offset "\n");
 if (i_modID == "null" || i_objID == "null") {
   ToolNet_ack("TN_PS_showObject has received null module or object parameters.");
   return;
 }
 // dereference ToolNetID
 string mode = "dontCloseLast";
 mdlToWorkIn = ToolNet_idmap_getModuleForID(i_modID, true, mode);
 current = mdlToWorkIn;
 if (filtering mdlToWorkIn) {
   filtering off;
 }
 objToShow = ToolNet_idmap_getObjectForID(mdlToWorkIn, i_objID);
 // no object, do nothing
 if (null(objToShow)){
   ToolNet_ack ("showObject: failed - could not find object\n");
   return;
 }
 current = objToShow;
 string absNum = i_objID;
 string attName = i_attribute;
 string offSet = i_offset;
 if (attName == "null")
   attName = null;
 // create filter
 Filter f1 = (attribute "Absolute Number" == absNum);
 // show object
 if (canModify(mdlToWorkIn)) {
   setSelection(objToShow);
 }
 ...
 set f1;
 filtering on;
 // show ancestors ?
 ancestors(false);
 // refresh module window
 refresh mdlToWorkIn;
 // bring module window to front
 ToolNet_moduleWinToFront(mdlToWorkIn);
 // ToolNet window
 if (ToolNet_blnWindow) {
     ToolNet_window_updateModule(ToolNet_window_dbMain, current());
     ToolNet_window_updateObject(ToolNet_window_dbMain, current(), current());
     ToolNet_window_updateAttribute(ToolNet_window_dbMain, current(), current(),
                                    attName);
     // window is hidden and we have non interaction modus
     if (! showing(ToolNet_window_dbMain)) {
       //realize(ToolNet_window_dbMain);
     }
 }
 ToolNet_ipc_sendVoidReturn( i_act );
}
```

Appendix B. A Prototype Walkthrough

B.1. Preconditions

For designing the prototype ServiceAssembly, the requirements outlined in Section 6.2 have to be met. For running the prototype, a valid license for the commercial application DOORS is needed (evaluation licenses are available from Telelogic on request).

For DOORS, it is assumed that a database with at least one formal module containing one or more objects is loaded. For this, the demo database was used throughout development and for walking through the prototype use case.

B.2. Designtime

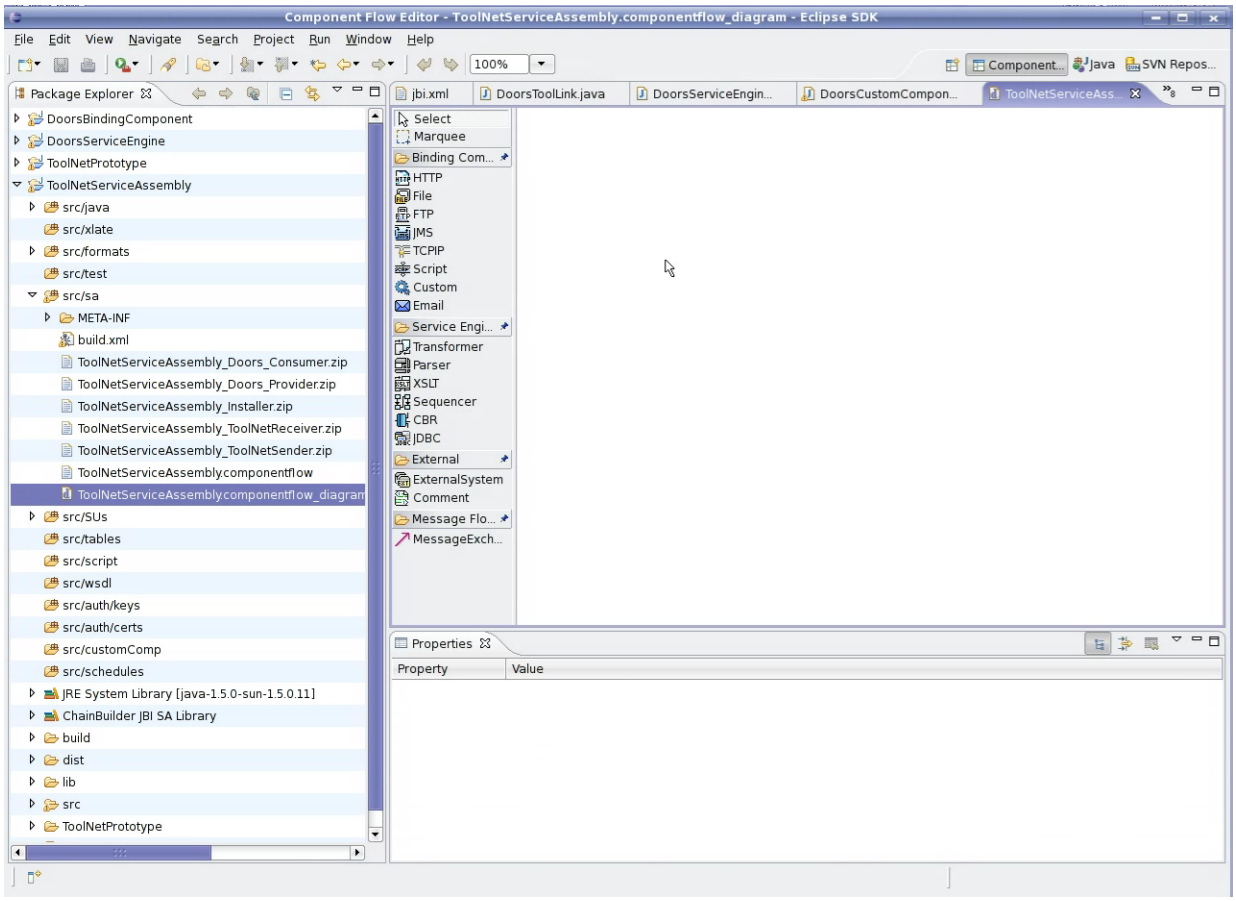


Figure B.1: Designing the prototype ServiceAssembly in the ChainbuilderESB IDE

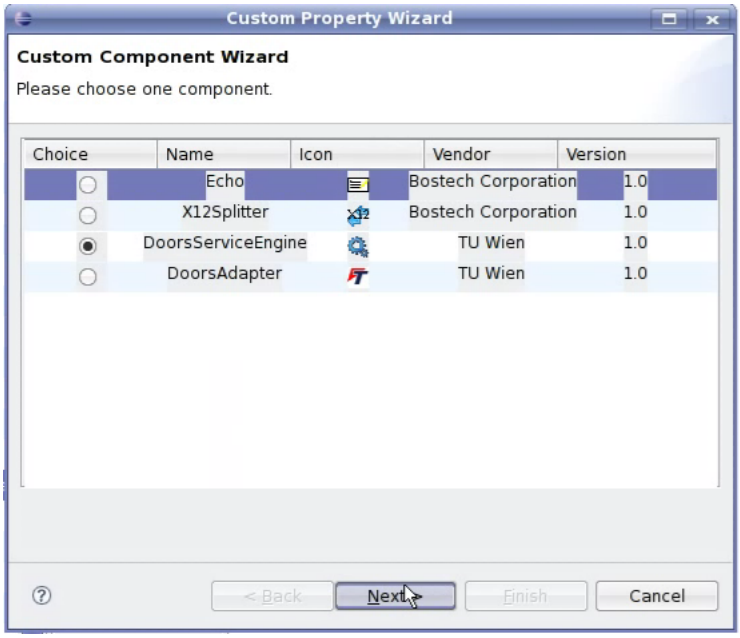


Figure B.2: Adding a new DOORS ServiceEngine

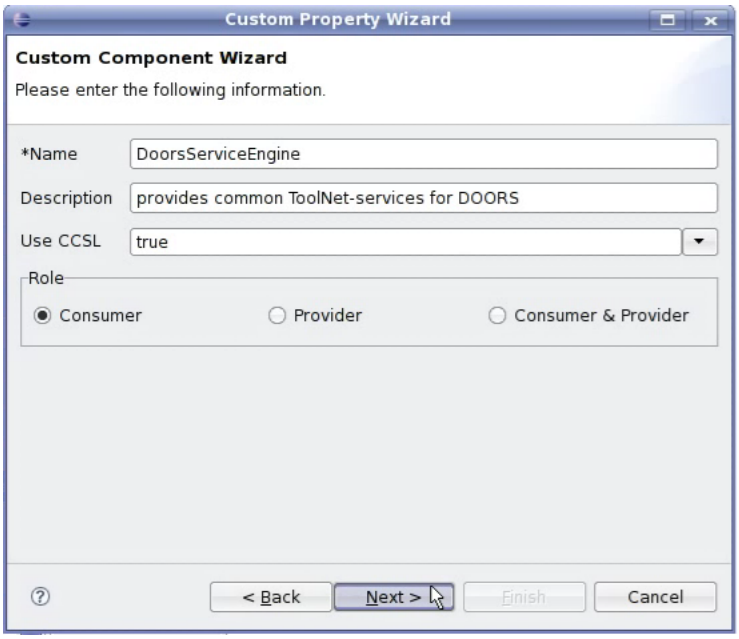


Figure B.3: Configuring the DOORS ServiceEngine as a Consumer

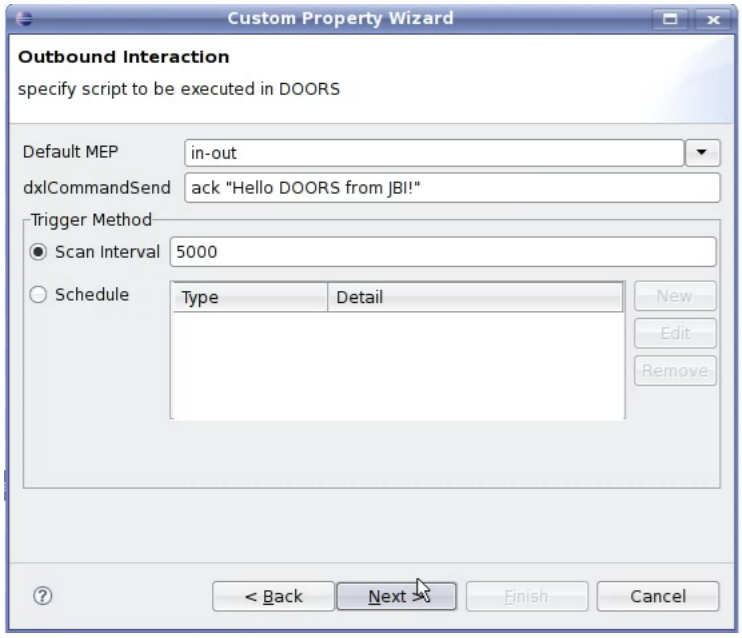


Figure B.4: Configuring the DOORS ServiceEngine's MessageExchangePattern

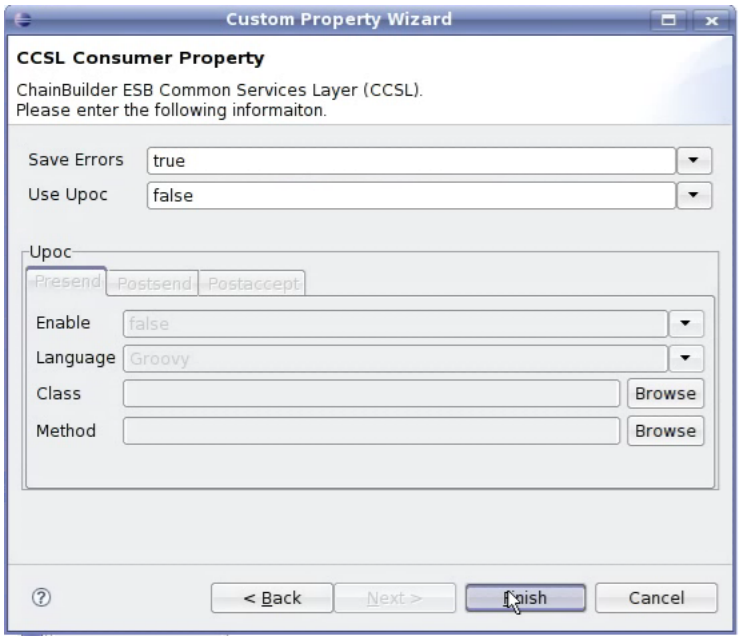


Figure B.5: Configuring the Chainbuilder helper library

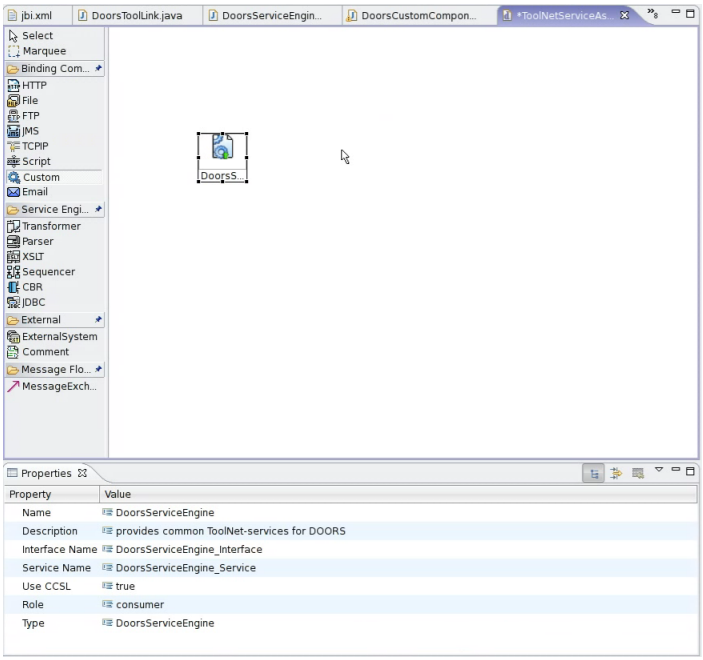


Figure B.6: The new ServiceEngine is displayed in the design view

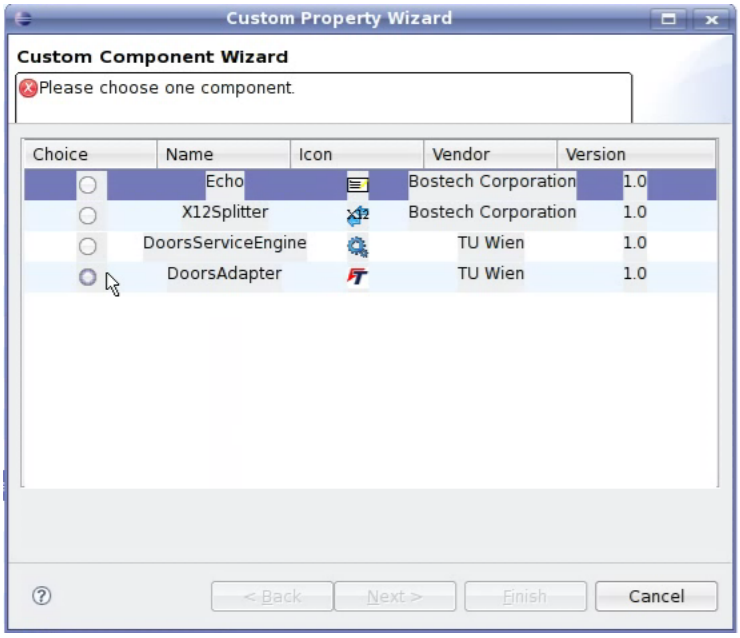


Figure B.7: Adding a new DOORS BindingComponent for sending requests to DOORS

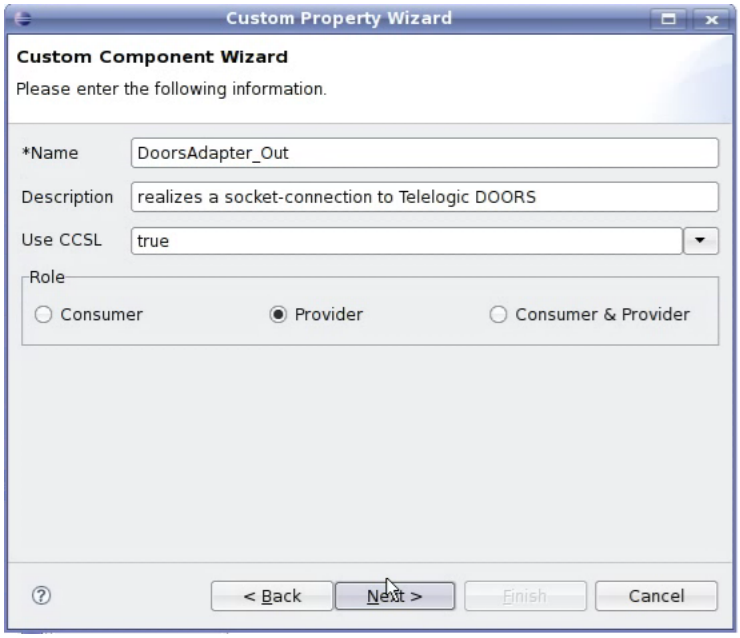


Figure B.8: Configuring the DOORS BindingComponent as a Provider

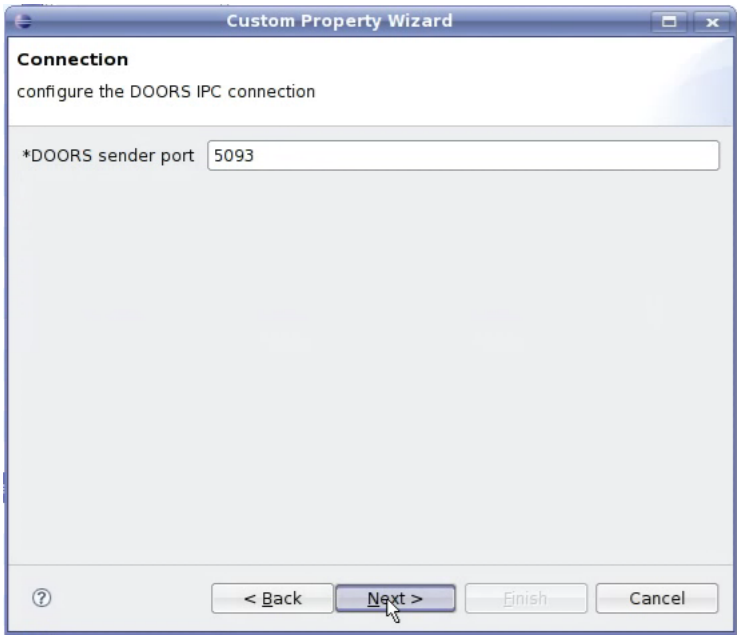


Figure B.9: Setting the DOORS sender port

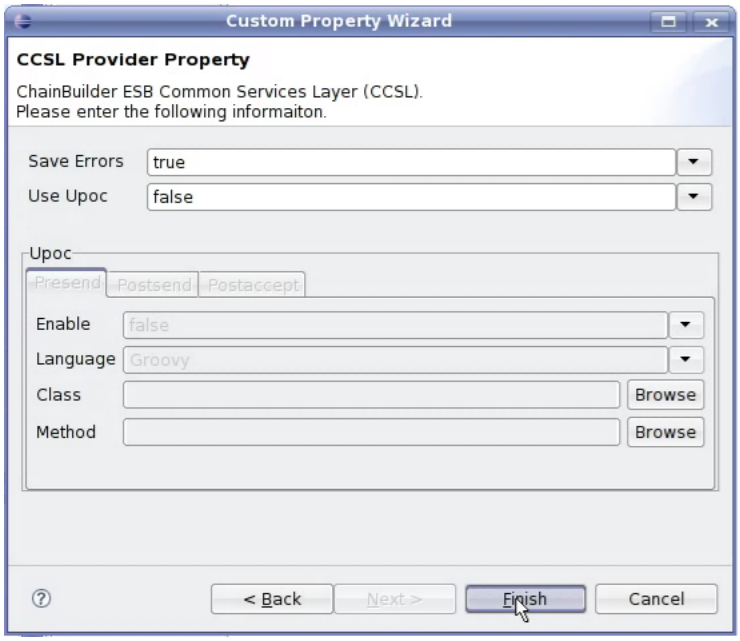


Figure B.10: Configuring the ChainBuilder helper library

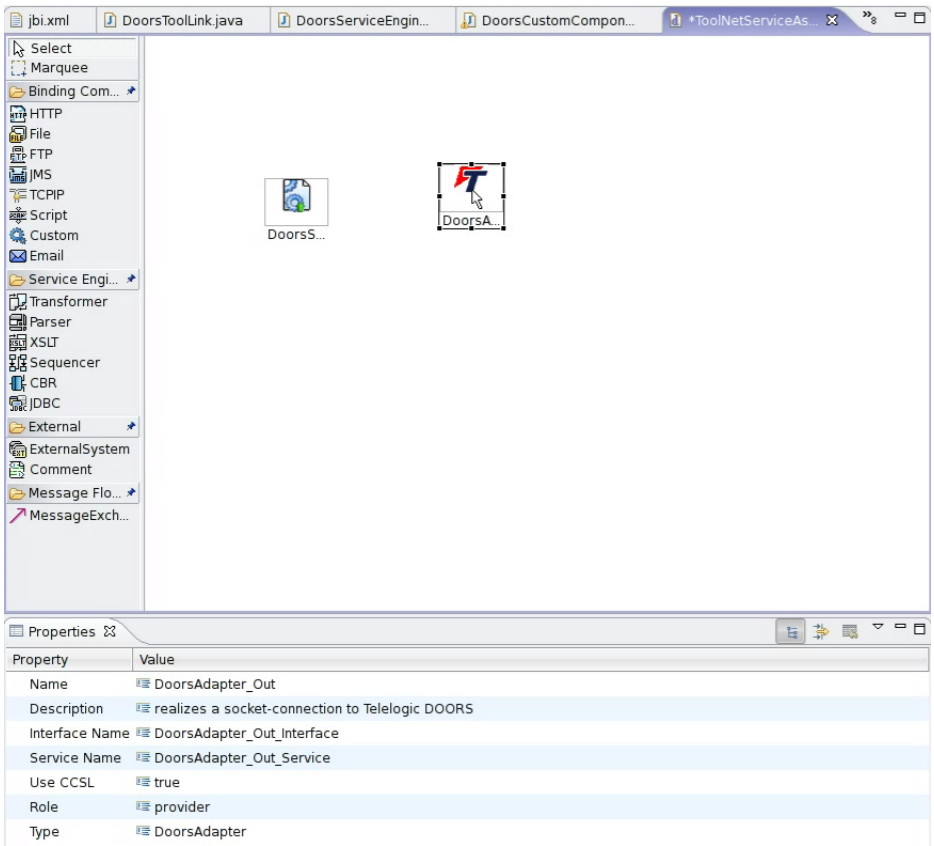


Figure B.11: The new BindingComponent is displayed in the design view

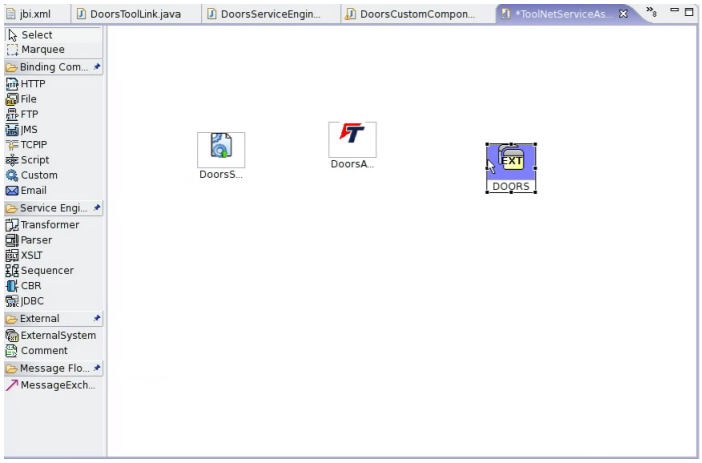


Figure B.12: Adding an external endpoint

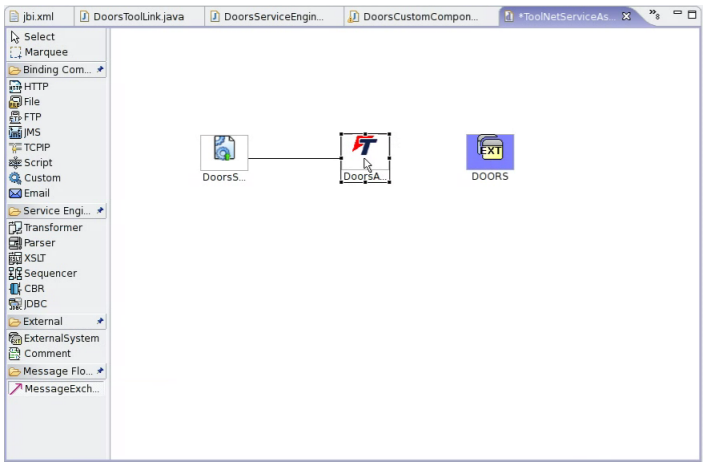


Figure B.13: Configuring the MessageExchange from ServiceEngine to BindingComponent

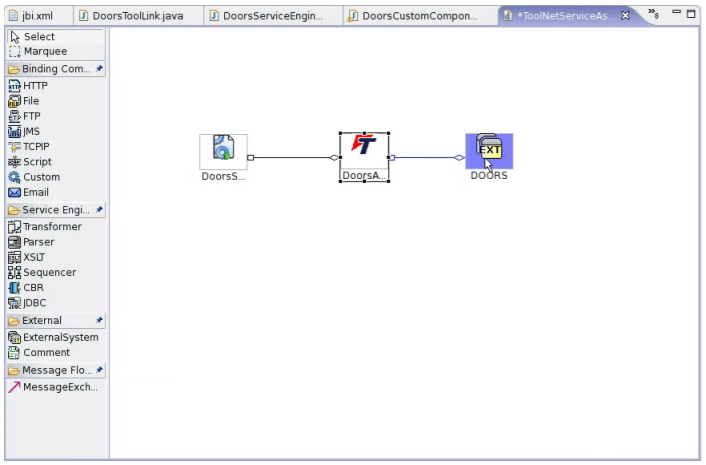


Figure B.14: Configuring the outgoing MessageExchange from BindingComponent to DOORS

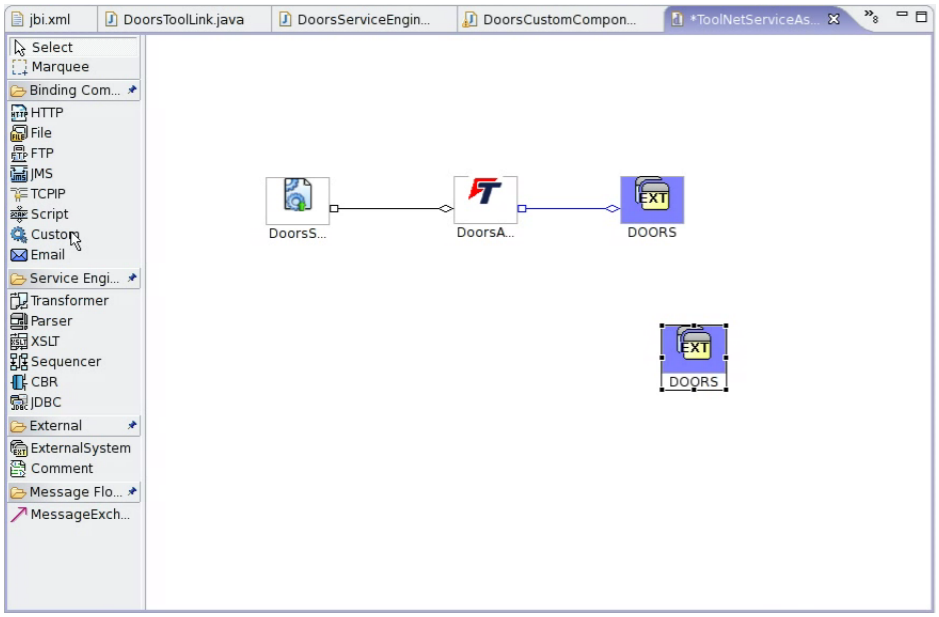


Figure B.15: Adding an incoming DOORS connection

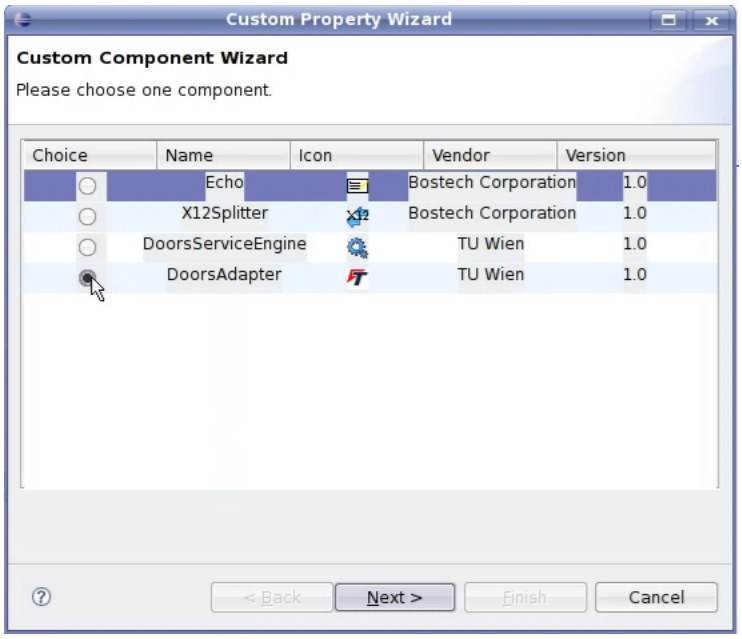


Figure B.16: Adding a new BindingComponent for handling incoming requests from DOORS

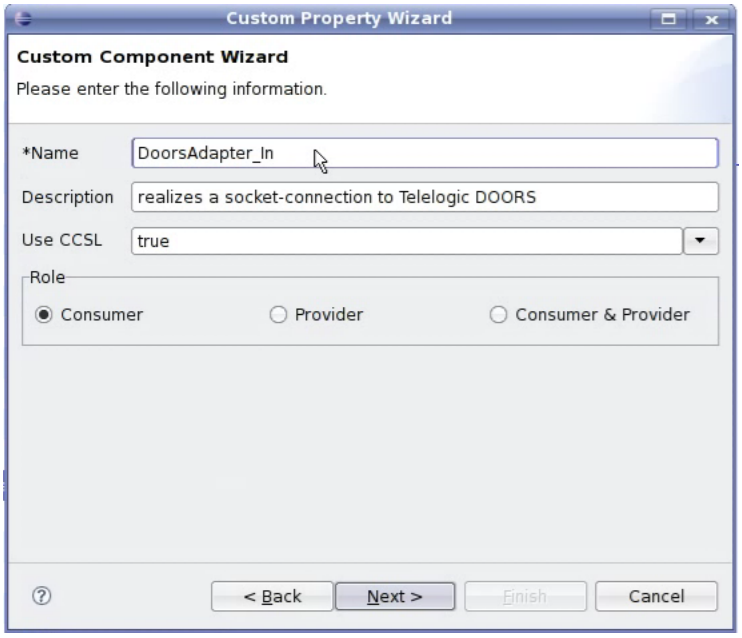


Figure B.17: Configuring the BindingComponent as Consumer

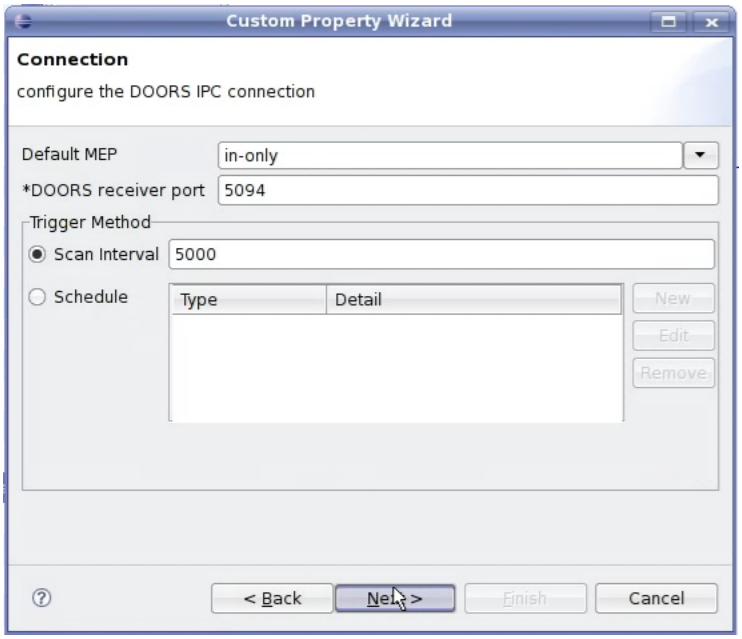


Figure B.18: Setting the BindingComponent's MessageExchangePattern and receiver port

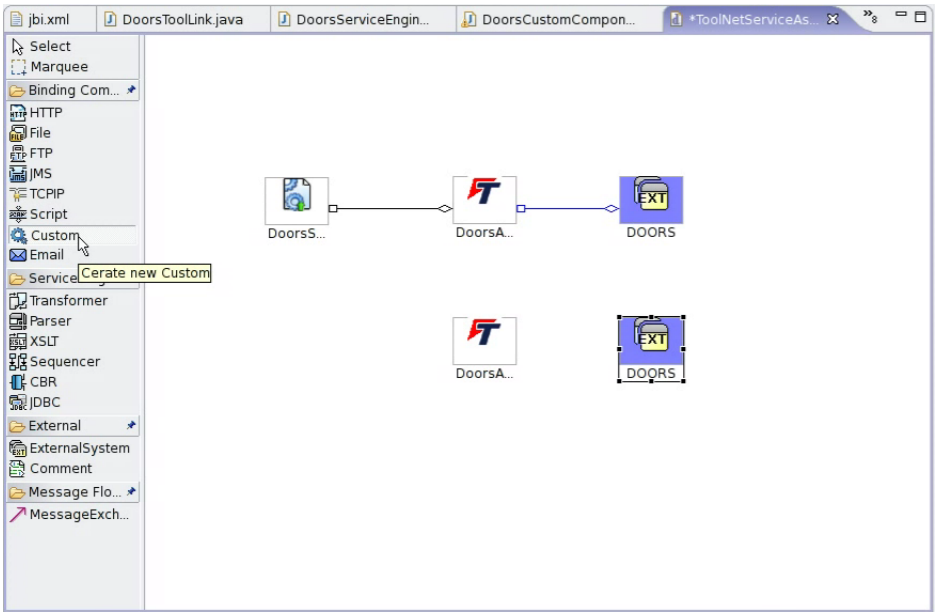


Figure B.19: The Consumer BindingComponent is displayed in the editor

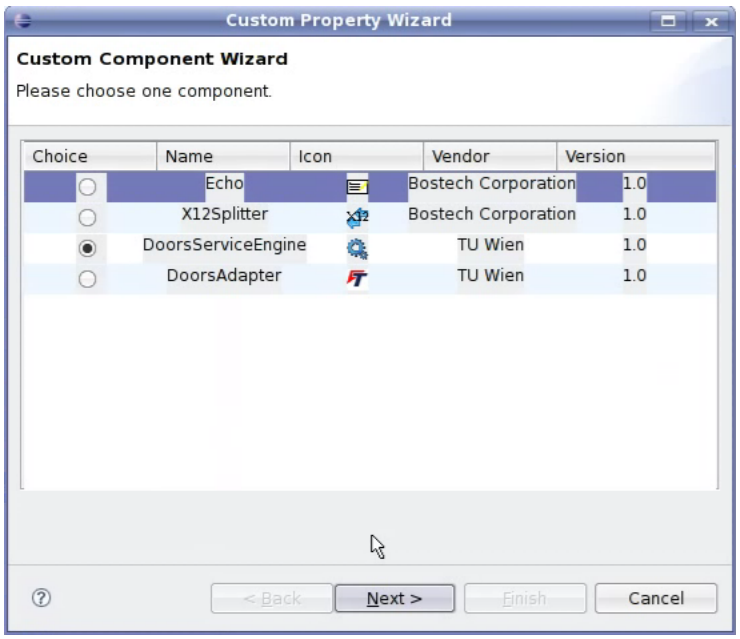


Figure B.20: Adding a ServiceEngine to process input from DOORS

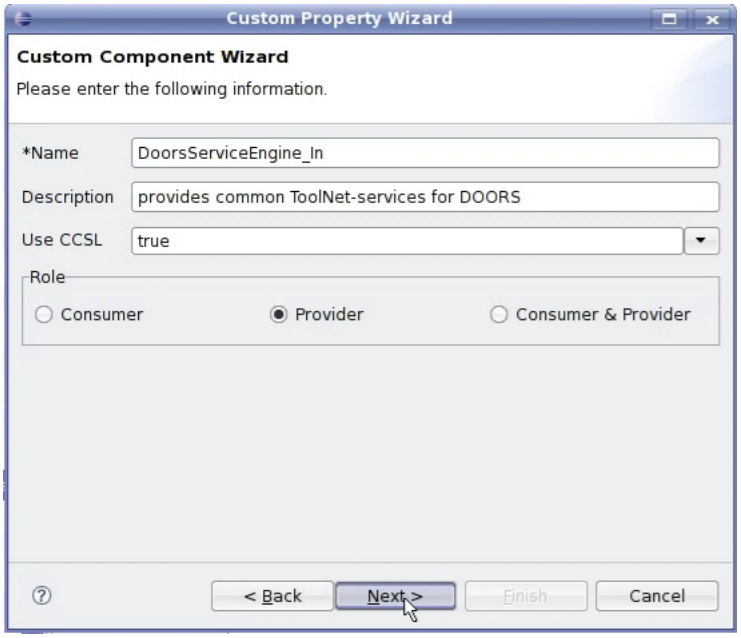


Figure B.21: Configuring the DOORS ServiceEngine as a Provider

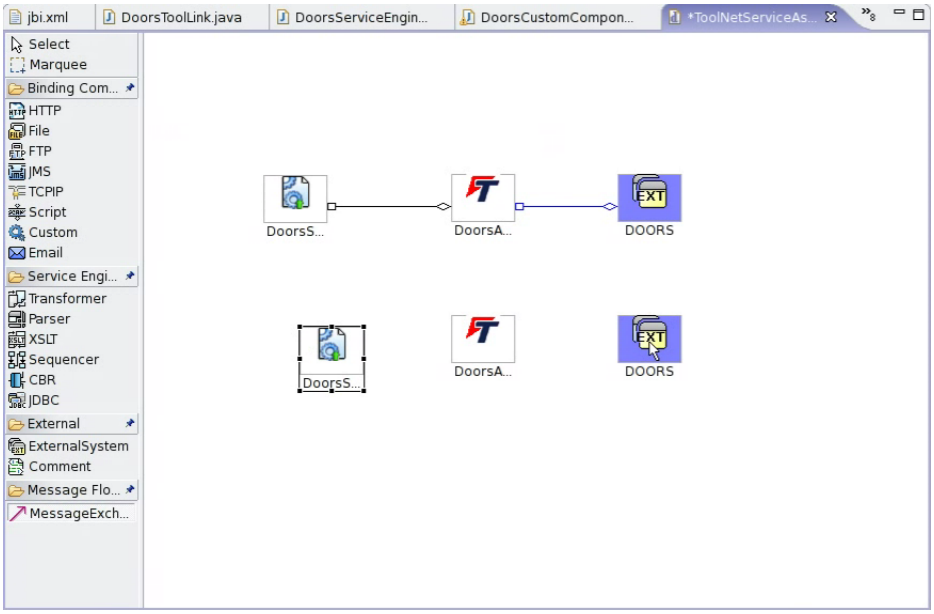


Figure B.22: The Provider ServiceEngine is displayed in the editor

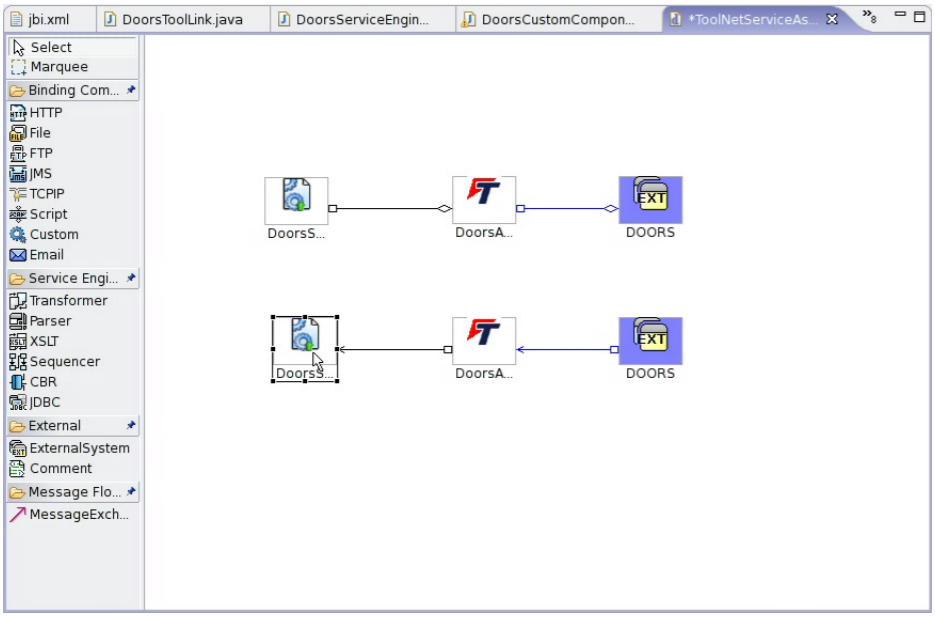


Figure B.23: Configuring the incoming message flow

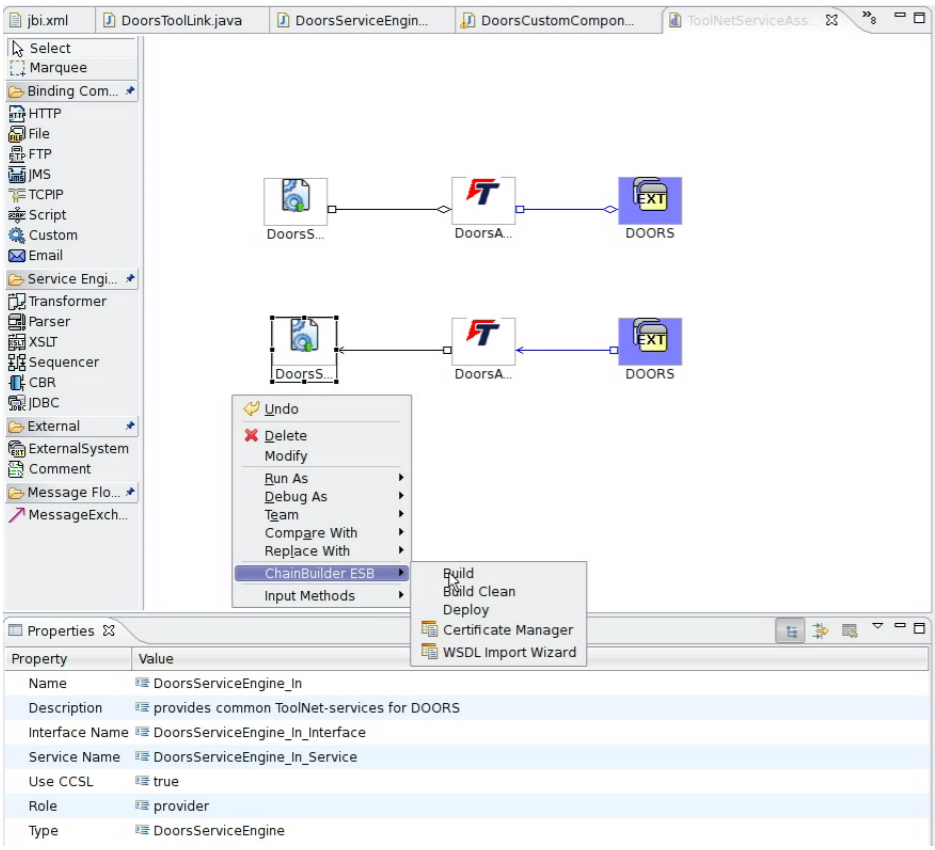


Figure B.24: Building the ServiceAssembly

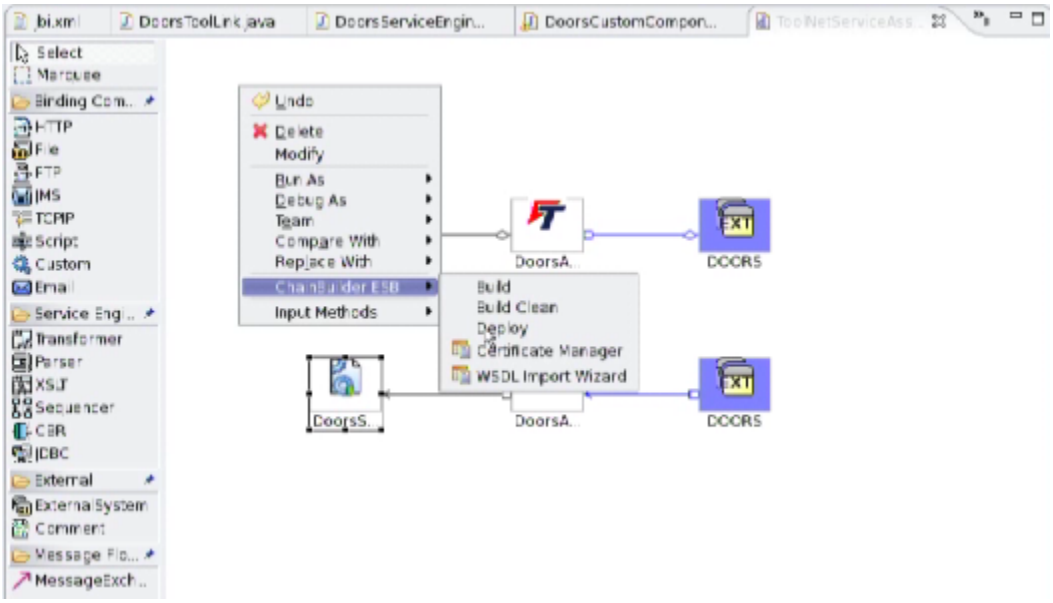


Figure B.25: Deploying the ServiceAssembly

B.3. Runtime

In the shell, set up the Chainbuilder environment:

```
# . set_cbesb.sh
```

Start the ServiceAssembly with the command:

```
# cbesb_run ToolNetServiceAssembly
```

The ServiceAssembly has to be located in ChainBuilder's installation directory under `runtimes/test`, which is the default location for ServiceAssemblies developed with the ChainBuilder IDE.

Wait until ServiceMix has started and finished installing the ServieAssembly:

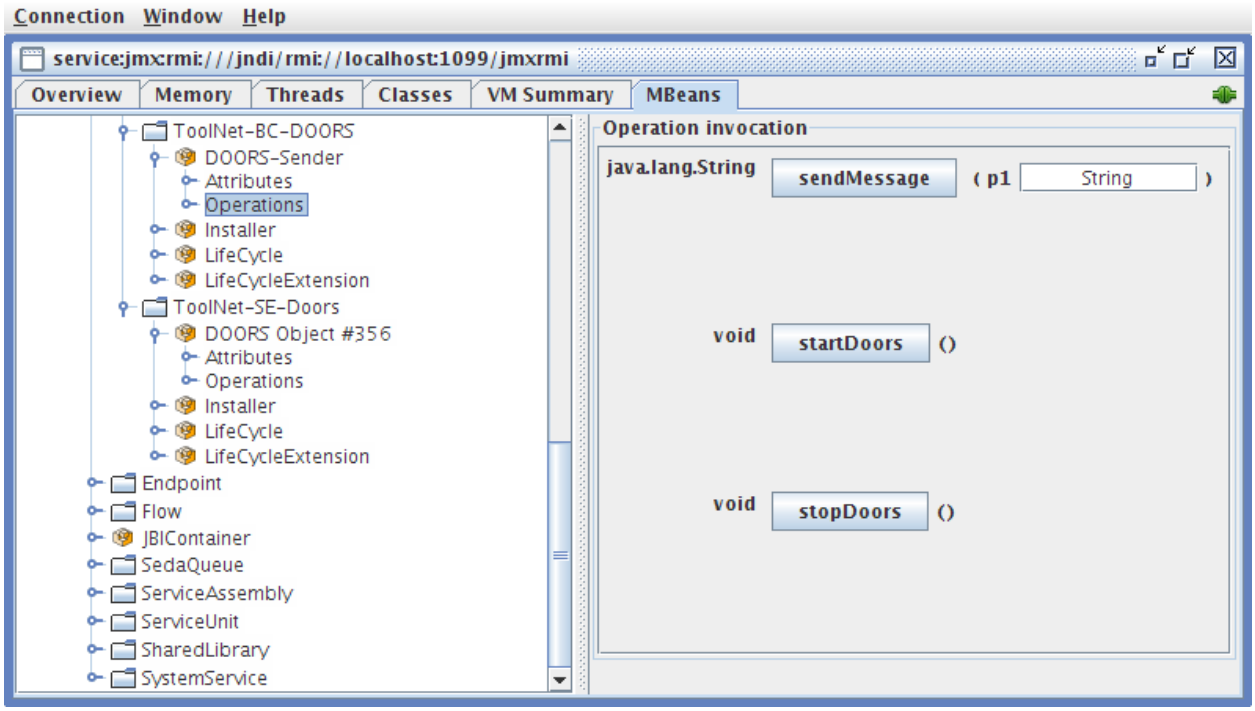
```
Starting Apache ServiceMix ESB: 3.2.1
Loading Apache ServiceMix from file: servicemix.xml
INFO - JBIContainer - ServiceMix 3.2.1 JBI Container (ServiceMix) is starting
...
INFO  - ServiceAssemblyLifeCycle      - Starting service assembly: ToolNetServiceAssembly
INFO  - ServiceUnitLifeCycle          - Initializing service unit: ToolNetServiceAssembly_DoorsServiceEngine
INFO  - ServiceUnitLifeCycle          - Initializing service unit: ToolNetServiceAssembly_DoorsAdapter_Out
INFO  - ServiceUnitLifeCycle          - Initializing service unit: ToolNetServiceAssembly_DoorsAdapter_In
INFO  - ServiceUnitLifeCycle          - Initializing service unit: ToolNetServiceAssembly_DoorsServiceEngine_In
INFO  - ServiceUnitLifeCycle          - Initializing service unit: ToolNetServiceAssembly_Installer
INFO  - ServiceUnitLifeCycle          - Starting service unit: ToolNetServiceAssembly_DoorsServiceEngine
INFO  - DoorsServiceEngineConsumerHandler - doStart()
INFO  - ServiceUnitLifeCycle          - Starting service unit: ToolNetServiceAssembly_DoorsAdapter_Out
INFO  - ServiceUnitLifeCycle          - Starting service unit: ToolNetServiceAssembly_DoorsAdapter_In
ERROR - DoorsConsumerListener         - Got invalid port: null - using default port 5094
INFO  - DoorsConsumerHandler          - ConsumerHandler started.
INFO  - ServiceUnitLifeCycle          - Starting service unit: ToolNetServiceAssembly_DoorsServiceEngine_In
INFO  - ServiceUnitLifeCycle          - Starting service unit: ToolNetServiceAssembly_Installer
INFO  - AutoDeploymentService         - Directory: deploy: Finished installation of archive:  ToolNetServiceAssembly.zip
```

Verify that all ServiceUnits have been started correctly and the DoorsConsumerListener Thread has started listening on the incoming DXL port configured.

Then open jconsole in a second Terminal, connecting to the local ServiceMix instance:

```
# jconsole service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi &
```

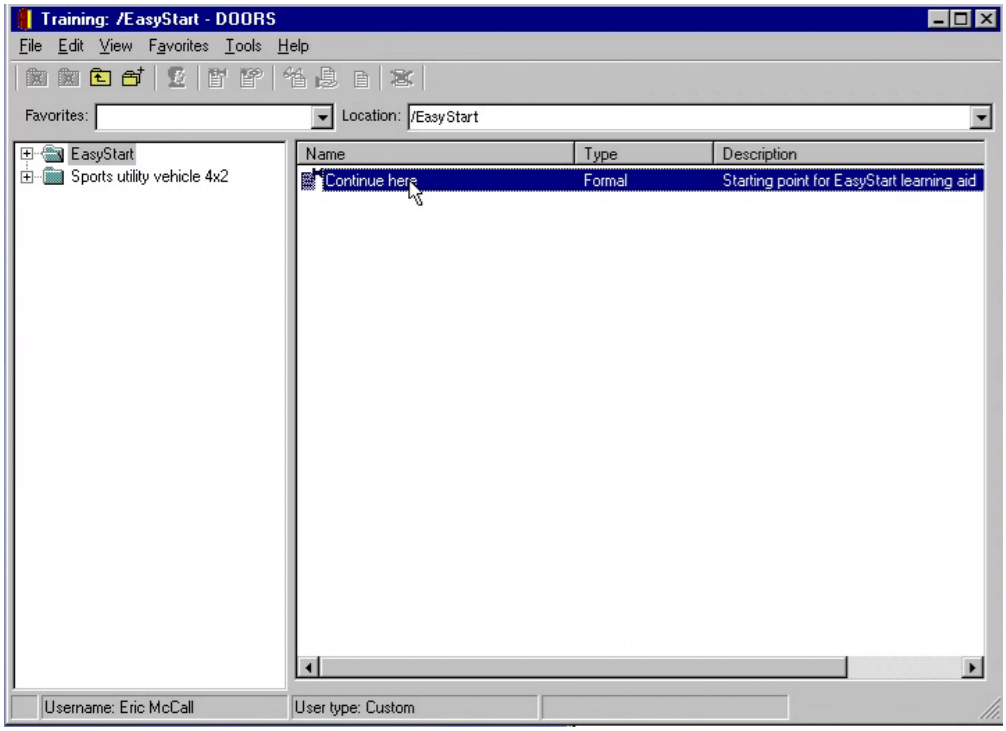
In JConsole, switch to the MBeans-tab and navigate to ServiceMix#Components#Toolnet-BC-Doors, then open the treeview and select DOORS-Sender#Operations. In the right view, Operation invocation, click on Start DOORS ❶, as shown in the screenshot below:



The application DOORS is started by the DOORS MBean ❶, and the DOORS-side ToolNet-Adaptor is initialized automatically when DOORS starts up. The Adaptor immediately connects to the prototype using a socket connection provided by the DOORS API and send the command "toolStarted()" for acknowledging successful initialization of the ToolNetAdapter. The response is received by the DOORS BindingComponent ❷ and handed over to the DoorsServiceEngine ❸, where it is interpreted accordingly ❹ (see log output below):

```
DEBUG - DoorsConfiguration               - Starting DOORS over JMX with command: /home/grexe/apps/doors71/bin/doors7 ❶
INFO  - DoorsConsumerListener            - Received input from DOORS: ❷
INFO  - DoorsConsumerListener            - org.toolnet.core.model.other.ILocalToolNet:toolStarted()
DEBUG - DoorsConsumerListener              - createInbound, DefaultMEP: http://www.w3.org/2004/08/wsdl/in-only ❸
DEBUG - DoorsServiceEngineComponent        - Received exchange: status: Active, role: provider
DEBUG - DoorsServiceEngineProviderProcessor - Received In Message:
 org.toolnet.core.model.other.ILocalToolNet:toolStarted()
DEBUG - DoorsServiceEngineProviderProcessor - parsing request ❹
DEBUG - DoorsServiceEngineProviderProcessor - DOORS Adapter started successfully
```

In the DOORS main window, open the formal module Easy Start and select the submodule Continue here by doubleclicking the item, as illustrated below:



A new window opens, showing the contents of the formal module. Now select any object (text) in the right view, then invoke the menu option ToolNet#Set object as source (notice that this menu is only available when a formal module is open, it is not visible in the main window at startup):

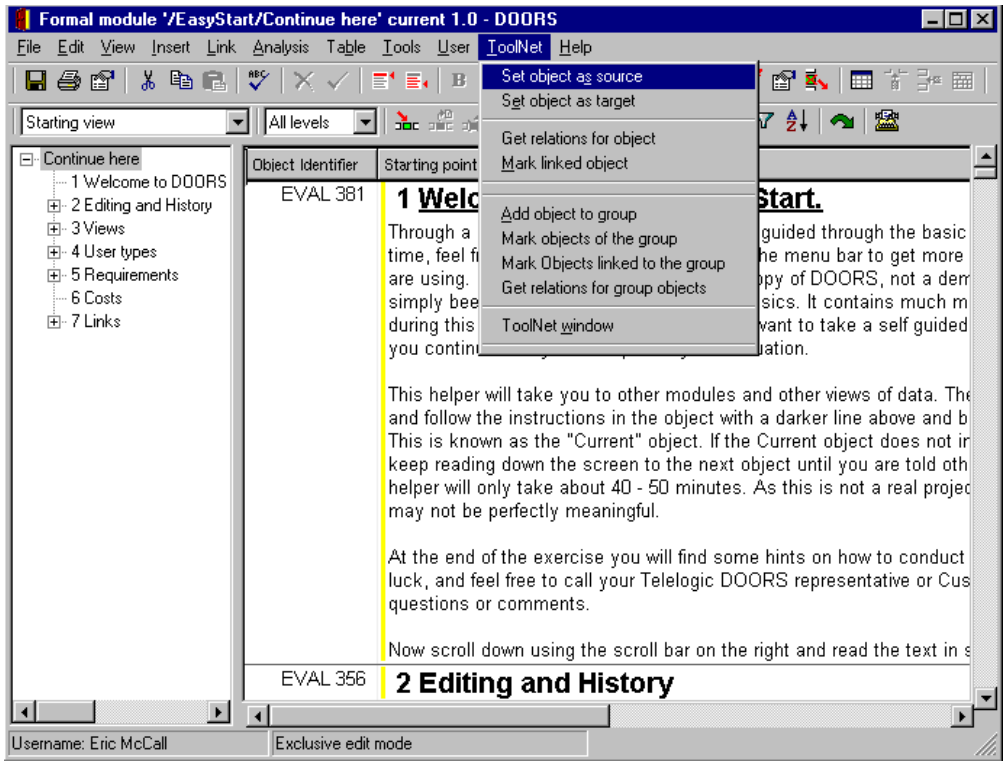


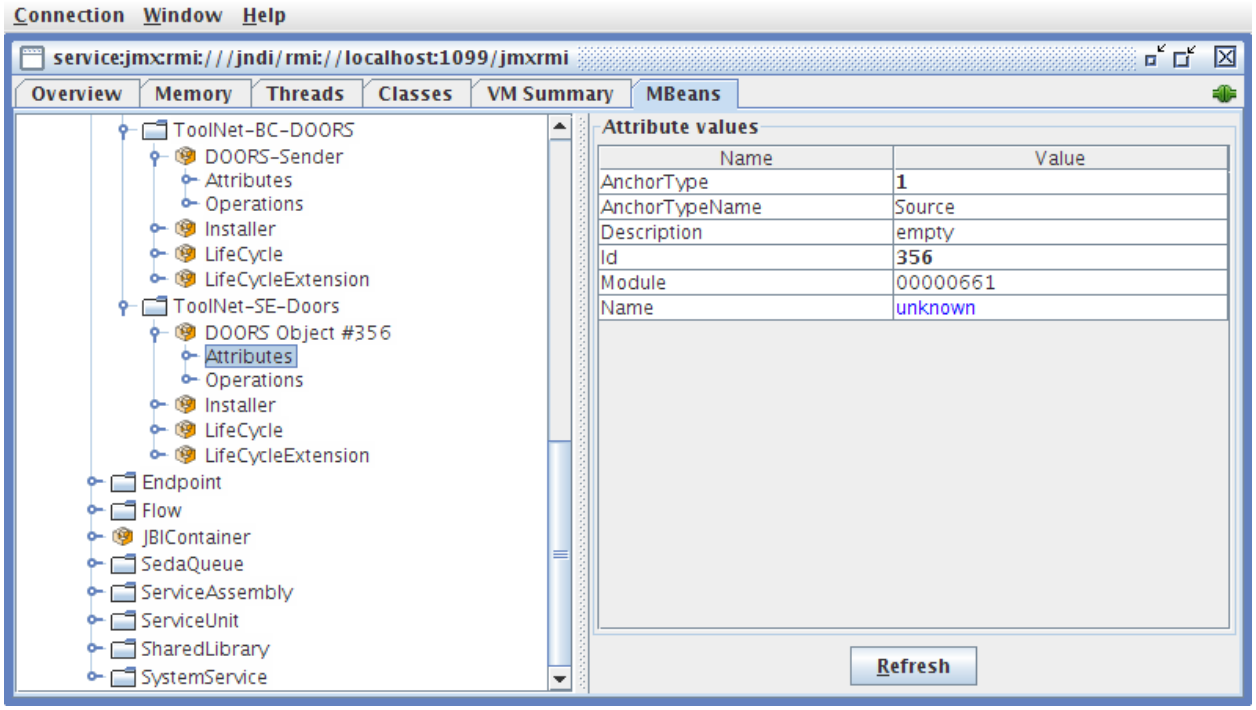
Figure B.26: Doors Source

Invoking the menu operation triggers the corresponding ToolNet-DXL-script of the DOORS ToolNet-Adapter, which sends a ToolNet-message over the IPC channel, containing the module ID, object ID and link type, along with the command that was invoked. This message is received by the DOORS BindingComponent on the prototype end ❶ (see the log below) and converted into a NormalizedMessage, which is sent to the ServiceEngine for further processing ❷. When the message is received by the DoorsServiceEngine ❸, it is propagated to the Provider ❹ which parses the request and associated arguments ❺. This information is then used to create a new MBean to represent the requested link to the DOORS Object ❻.

```
INFO  - DoorsConsumerListener          - Received input from DOORS: ❶
DEBUG - DoorsConsumerListener          - org.toolnet.core.model.services.IRelationCreation:addAnchor((id)
["00000661","356","__NULL__","__NULL__"],(AddAsType)"1")
DEBUG - DoorsConsumerListener          - createInbound, DefaultMEP :http://www.w3.org/2004/08/wsdl/in-only ❷
DEBUG - DoorsConsumerListener          - Consumer endpoint service=...
DEBUG - DoorsConsumerListener          - Got target endpoint...
DEBUG - DoorsServiceEngineComponent    - Received exchange: status: Active, role: provider ❸
INFO  - DoorsServiceEngineProviderProcessor - Received In Message: ❹
org.toolnet.core.model.services.IRelationCreation:addAnchor((id)["00000661","356","__NULL__","__NULL__"],
(AddAsType)"1")

DEBUG - DoorsServiceEngineProviderProcessor - parsing request:
 org.toolnet.core.model.services.IRelationCreation:addAnchor((id)["00000661","356","__NULL__","__NULL__"],
(AddAsType)"1")
DEBUG - DoorsServiceEngineProviderProcessor - DOORS Adapter addAnchor requested: ❺
DEBUG - DoorsServiceEngineProviderProcessor - parsed module: 00000661
DEBUG - DoorsServiceEngineProviderProcessor - parsed ID: 356
DEBUG - DoorsServiceEngineProviderProcessor - setting up DOORS Object MBean for ID 356 ❻
DEBUG - DoorsServiceEngineProviderProcessor - Registering DOORS Object MBean
 'org.apache.servicemix:ContainerName=ServiceMix,Type=Component,Name=ToolNet-SE-Doors,SubType=DOORS Object
 #356'
DEBUG - DoorsObject                    - init ObjectMBean for ConsumerHandler (ID=356, type=1)
DEBUG - DoorsServiceEngineProviderProcessor - successfully registered MBean
 org.apache.servicemix:ContainerName=ServiceMix,Type=Component,Name=ToolNet-SE-Doors,SubType=DOORS Object
 #356
DEBUG - DoorsServiceEngineProviderProcessor - DONE with MessageExchange
```

The new MBean is displayed as "DOORS Object #" including the ObjectID as received from DOORS. The screenshot below shows the updated MBean view in JConsole:



Each DOORS Object MBean also includes an operation to highlight the corresponding Object in DOORS. In the JMX console, navigate to ServiceMix#Components#Toolnet-SE-DOORS. Go to Operations and click on the highlight button:

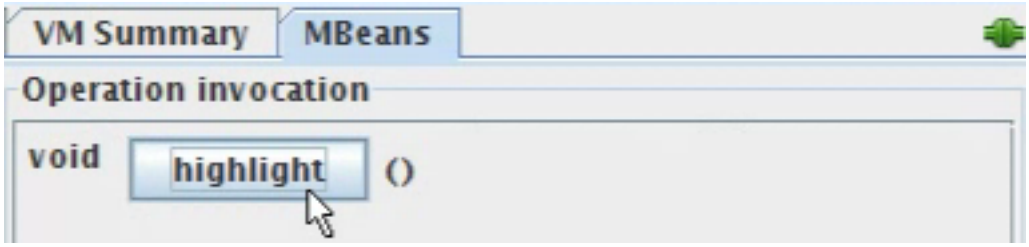


Figure B.27: Highlighting a linked Object in DOORS from the prototype using JConsole

When the method is invoked on the DoorsMBean, it calls the DoorsServiceEngine's `highlightObject()`-method ❶ that sends a DXL-script to the DoorsBindingComponent. Upon receiving the request message ❷, the DOORS BindingComponent opens a socket connection to the DXL server in DOORS ❸ and sends the DXL over the wire ❹. The ToolNet-Adapter in DOORS interprets the script and executes the commands accordingly, changing the active view to show only the object specified in the script. Then it sends an acknowledgement back to the prototype to inform about the successful operation, which is received on the prototype end ❺, again by the DoorsBindingComponent, and parsed accordingly by the DoorsServiceEngine ❻, which registers the success response ❼.

```
DEBUG - DoorsServiceEngineConsumerHandler - Got message over JMX... ❶
DEBUG - DoorsServiceEngineConsumerHandler - Sending to the NMR...
DEBUG - DoorsServiceEngineConsumerHandler - create IN-ONLY
DEBUG - DoorsServiceEngineConsumerHandler - Normalized Message:
#include <addins/ToolNet/ToolNet_PresentationService.inc>;
ToolNet_IPresentation_showObject("00000661","356","null","null","HIGHLIGHT_OBJECT")
DEBUG - DoorsServiceEngineConsumerHandler - message sent successfully to NMR.
DEBUG - DoorsComponent                - Received exchange: status: Active, role: provider
```

```
DEBUG - DoorsProviderProcessor        - Received in message. ❷
WARN  - DoorsProviderProcessor        - using default port: 5093
DEBUG - DoorsProviderProcessor        - Connecting to DOORS @5093 ❸
INFO  - DoorsProviderProcessor        - Successfully Connected to DOORS.
DEBUG - DoorsProviderProcessor        - sending DXL: ... ❹
INFO  - DoorsConsumerListener         - Received input from DOORS: ❺
DEBUG - DoorsConsumerListener          - return HIGHLIGHT_OBJECT
DEBUG - DoorsConsumerListener          - createInbound, DefaultMEP :http://www.w3.org/2004/08/wsdl/in-only
...
INFO  - DoorsServiceEngineProviderProcessor - Received In Message: return HIGHLIGHT_OBJECT
DEBUG - DoorsServiceEngineProviderProcessor - parsing request: return HIGHLIGHT_OBJECT ❻
DEBUG - DoorsServiceEngineProviderProcessor - DOORS-Adapter successfully invoked operation: HIGHLIGHT_OBJECT ❼
DEBUG - DoorsProviderProcessor        - Command sent to DOORS successfully.
```

To reset the view in DOORS, click on the active filter-icon in the Toolbar (2nd row, left to the sorting icon "A-Z"). You can now link more Objects and invoke the highlight-operation from the additional MBeans that come up in the JConsole MBeans-view, following the previous steps again.

Glossary

List of Terms and Abbreviations

BPEL (Business Process Execution Language)	an XML language for defining business work flows, often used in ESB See Also BPM.
BPM (Business Process Management)	a disipline that covers the analysis, design and optimization of business processes in an enterprise, working together with enterprise architects and integration designers
COTS (commercial of the shelf software)	pre-packaged software acquired from an external vendor; mostly closed-source applications using proprietary interfaces and data formats, optionally provide an API, making the software accessible to other applications See Also ToolNet.
CORBA (Common Request Broker Architecture)	an open, vendor-independent specification defined by the OMG consortium; defines an architecture and infrastructure to provide interoperability between distributed, heterogeneous applications using the standard protocol IIOP.
COM (Component Object Model)	a proprietary component API introduced by Microsoft in Windows during the 1990s, later became COM+ and DCOM, then ActiveX, now superseded by .NET
Composite Application	an application that is composed of independent, often service-oriented applications; unlike past component approaches, composite applications are designed for distributed interoperability using abstract interfaces, and allow for dynamic and spontaneous integration by integration designers or even normal end users. See Also COM.
CRM (Customer Relationship Management)	business discipline for integrating customer information with other relevant data, like past and present inquiries (mail), orders or appointments; this is a classical candidate for integration in the enterprise domain
DI (Dependency Injection)	The term was coined by Martin Fowler to describe a pattern first called *inversion of control*, which can be seen as an implementation of the *Hollywood-principle* (“don't call us, we'll call you”). Following this method, dependencies are not directly resolved by the application at designtime, but dynamically *injected* into placeholders, e.g. configuration classes, during runtime, using a framework and associated configuration, e.g. Spring. This allows late binding of data sources or other needed resources, without tying the implementation to specific dependencies, allowing easier unit testing and adapting to changing requirements.
DocBook	an open XML-based standard for writing technical documentation, books and articles, similar to (La)TeX but with the advantages of using a well-defined XML-format and flexible XSL-stylesheets. DocBook, now at version 5, helps the author to focus on the semantic aspect of writing by providing a rich set of notations and automates formatting for various output-formats by using suit-

	able stylesheets (such as OpenDocument, HTML, FO/PDF or PS). Also translation to legacy documentation formats such as Word (DOC) or TeX is possible.
DXL (DOORS eXtension Language)	a scripting language to access the *DOORS* API from external applications, used for integrating DOORS into *ToolNet* See Also DOORS.
DOORS (Dynamic Object Oriented Requirements System)	a widely used commercial application for requirements-tracing developed by Telelogic, used as a prime example for integration of *COTS* tools with *ToolNet* See Also COTS software, ToolNet.
Eclipse	an extensible and cross-platform `open source IDE`[1] and application platform originally developed by IBM. Based on Java and OSGi, it provides a flexible plugin-based extension architecture to facilitate adoption of the IDE for a wide array of usage scenarios in various software and system engineering domains. See Also NetBeans, OSGi, Rich Client Platform.
EAI (Enterprise Application Integration)	describes the general domain of integrating applications, mostly legacy backend systems, in an enterprise environment to allow reusing existing software together with new technologies, such as *SOA*, and newly added applications
EJB (Enterprise Java Beans)	a *JEE* component standard tailored to enterprise needs, including support for transactions and distributed communication, e.g., using *Web Services*
ESB (Enterprise Service Bus)	an integration solution providing a flexible communication backbone that supports conversion to and from multiple protocols used by existing applications to be connected. See Also Java Business Integration.
EDA (Event Driven Architecture)	Whereas in an *SOA*, service calls are issued by services themselves in an imparative fashion, the principle notion of an event-driven architecture is the Event, which can be generated by an external source, a connected application, or triggered as a result of another Event. This indirect communication enables developers to build more flexible and dynamic real-world solutions that can intelligently process massive requests in an automated way. In practice, both approaches complement each other and are often used together. See Also SOA.
ETL (Extract, Transfer and Load)	a common integration pattern in the enterprise domain for connecting and synchronizing large data sources, mostly realized as nightly batch jobs in transnational organizations
IPC (Inter Process Communication)	a mechanism that allows separate applications to communicate with each other, typically using an OS-level facility like sockets (enabling distributed communication) or pipes (on UNIX)
JBI (Java Business Integration)	a specification by Sun (JSR 208) that defines a standards-based service-oriented integration framework based on Java technology that can also incorporate non-Java applications. See Also Service Component Architecture.

JCA (Java Connector Architecture)	a specification by Sun (JSR 16) that defines a standard architecture and interface contracts for integrating existing legacy applications using `ResourceAdapters`. See Also Java Business Integration.
JEE (Java Enterprise Edition)	(called J2EE prior to JEE5) specifies a standard platform for developing enterprise applications in Java, defining several standard APIs for working with legacy systems (JCA), databases (JDBC), XML (SAX, JAXB, StAX), web services (JAX-WS), web interfaces (JSF) and remote applications (EJB, JSR 220). Several versions of the platform have been defined through the Java Community Process (JCP) in JSRs 58 (J2EE 1.3), 151 (J2EE 1.4), 244 (JEE 5), 316 (JEE 6). See Also Enterprise Java Beans.
JMX (Java Management Extensions)	defines a standard architecture, API and services for local and remote management and instrumentation of Java applications and management of the Java Virtual Machine through JSRs 3 (JMX Specification), 77 (Management for J2EE), 160 (remote management) and 255 (JMX 2.0). See Sun's `JMX Technology Homepage`[2] for more details.
JNA (Java Native Access)	an open source Java library that wraps access to native code over JNI by providing a proxy that implements a custom interface written by the user, thereby avoiding the effort necessary to write custom header files and stub classes for integrating non-Java code; see [JNA] See Also JNI.
JNI (Java Native Interface)	the standard way to access native (non-Java) code from Java, e.g. for integrating C libraries or legacy code; requires developers to write special headers and Java stubs – an easier way to integrate native doe is available with *JNA*
Mashup	The compounding (“mashing”) of two or more pieces of complementing web functionalities to create a powerful web application. This is usually achieved through the use of APIs. (taken from *A Quick Web 2.0 Glossary*, http://www.brownbatterystudios.com/sixthings/2006/02/24/a-quick-web-20-glossary/)
MOM (Message-Oriented Middleware Integration)	a kind of functional integration where systems are connected through message queues using proprietary messaging middleware (see [Trowbridge2004], p115)
NetBeans	a Java-based open source IDE developed by Sun. See Also Eclipse.
OLE (Object Linking and Embedding)	a Microsoft Windows standard for component-based application integration, allowing embedding (parts of) applications into other applications, e.g., a spreadsheet component into a text document, forming so called *compound* documents. Now mostly superseded by ActiveX.
OMG (Object Management Group)	an industry consortium that develops open standards for software developers and end users, sample OMG standards include UML or *CORBA*
CSA (Open Composite Services Architecture)	open collaboration led by OASIS Consortium to continue development of the *Service Component Architecture* (SCA)- and *Service Data Objects* (SDO)-specifications

	See Also Service Component Architecture.
OSGi (Open Services Gateway Initiative)	an industry-wide open component standard originally used in the embedded and automotive domain, but became widespread on the desktop with the adoption by Eclipse, using it as the basis for its plugin-framework; currently increasingly used in enterprise environments, replacing proprietary application server module architectures
OASIS (Organization for the Advancement of Structured Information Standards)	member consortium working on open (mostly document related) standards, see the OASIS web site http://www.oasis-open.org/ See Also WS-I.
REST (Representational State Transfer)	a distributed communication architecture that solely relies on HTTP for defining a set of common operations understood by all participating services, thereby avoiding the overhead in usual SOA implementations introduced by complex XML-based protocols and interface definitions See Also SOAP.
RCP (Rich Client Platform)	an application platform based on *Eclipse* and *SWT* to facilitate rapid creation of plugin-based portable applications with standard user interfaces. See Also Eclipse.
SCA (Service Component Architecture)	a standard for a service-oriented composite application framework originally developed by IBM, which is now being continued as an open standards effort called *Open Composite Services Architecture* under the umbrella of the `OASIS Consortium`[3]. See Also CSA.
SDO (Service Data Objects)	an XML-based standard for data-integration in heterogenous environments including enterprise information systems, web services and relational databases. Initially developed by IBM, now an open standard which is further developed as part of the *Open Composite Services Architecture*. See Also Service Component Architecture.
SOA (Service Oriented Architecture)	a design principle that uses design patterns, best practices and open interoperability standards to facilitate the realization of modular systems that expose their functionality as a set of independent *Services* described using a common interface schema (e.g., *WSDL*). A common facility (e.g., *UDDI*) allows other Services to query for registered Services and transparently invoke them (e.g., using SOAP), reusing existing functionality. SOA facilitates loose coupling between applications and maximizes reuse.
SOE (Service Oriented Enterprise (also Enterprise 2.0))	a marketing moniker for describing the adoption of Web 2.0 concepts in an enterprise environment, e.g. enterprise wikis or corporate application Mashup
SOI (Service Oriented Integration)	Unlike MOM, this integration form uses open, service-oriented standards to connect systems in a portable, loosely coupled way that is not bound to proprietary protocols or implementations. See Also ESB.
SOAP (Simple Object Access Protocol)	now only called SOAP, an open XML-based interoperability standard realizing remote communication among software components or Services, often used as part of an *SOA*

SWT (Standard Widget Toolkit)	an open source `UI-framework`[4] for Java developed by IBM mainly for use in the Eclipse IDE. Provides support for native widgets of the underlying operating system, unlike Swing (up to including JSE5 with limited support for native widgets in JSE6), the standard widget toolkit for Java.
ToolNet	a service-oriented framework for desktop application integration developed by EADS Germany for connecting heterogenous and *COTS* engineering tools to allow for data exchange and improved workflow for engineers See Also COTS.
UDDI (Universal Description, Discovery and Integration)	acts as a central directory in an SOA to manage registered services and handle queries (c.f. [Erl2004:80]) See Also SOA.
Web Service	application services exposed over the Web for distributed operation, using open standards for describing the interface (e.g., *WSDL*) and for communication (e.g. *SOAP*); recently, *REST*-based web services are emerging that rely only on HTTP methods like PUT, GET and DELETE, and avoid the overhead in using XML-based protocols and interface descriptions See Also SOA, REST.
WSDL (Web Service Description Language)	an XML-based standardized interface description format generally used in an *SOA* for defining Service interfaces See Also SOA.
WS-I (Web Services Interoperability)	open consortium that works on interoperability standards for *web services*, defining several profiles that represent levels of interoperability
WCF (Windows Communication Foundation)	a *Web Service* based distributed communication infrastructure for the Microsoft .NET framework See Also SOA.
XMI (XML Metadata Interchange)	an open, XML-based standard for interoperability between modeling applications, implementing a meta-model by the *OMG* See Also OMG.

References

[Altheide2002] Frank Altheide, Heiko Dörr, and Andy Schürr. “Requirements to a Framework for sustainable Integration of System Development Tools”. AFIS PC Chairs. 53-57. 2002.

[Altheide2003] Frank Altheide, Sven Dörfel, Heiko Dörr, and Jan Kanzleiter. “An Architecture for a Sustainable Tool Integration”. *Workshop on Tool Integration in System Development at ESEC/FSE 2003*. 29–32. 2003.

[Amsden2001] J. Amsden. “Levels of Integration: Five Ways You Can Integrate with the Eclipse Platform”. 2001.

[Anderson2000] K. Anderson, R. Taylor, and E. Whitehead. “Chimera: Hypermedia for Heterogeneous Software Development Environments”. 2000.

[Anderson1993] M.J. Anderson and B.D. Bird. “An evaluation of PCTE as a portable tool platform”. *Software Engineering Environments Conference, 1993. Proceedings*. 96-100. 1993.

[Apple2007] Apple. “Automator Programming Guide”. 2007.

[Apple1993] Apple. “Inside Macintosh: Interapplication Communication”. Addison-Wesley. 1st. 1993.

[Arnold1995] John E. Arnold. “Control integration: a briefly annotated bibliography”. *SIGSOFT Softw. Eng. Notes*. ACM. 20. 62–67. 1995.

[Arsanjani2005] Ali Arsanjani. “Toward a pattern language for Service-Oriented Architecture and Integration”. *IBM DeveloperWorks*. 2005.

[Balasubramanian2006] Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltán Molnár, and Ákos Lédeczi. “System Integration using Model-Driven Engineering”. 2006.

[Bandinelli1996] S. Bandinelli, E. Di Nitto, and A. Fuggetta. “Supporting cooperation in the SPADE-1 environment”. *Software Engineering, IEEE Transactions on*. 22. 841-865. 1996.

[BaoHorowitz1996] Yimin Bao and Ellis Horowitz. “A new approach to software tool interoperability”. ACM. 500–509. 1996.

[Barrett1996] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. “A framework for event-based software integration”. *ACM Trans. Softw. Eng. Methodol.*. ACM. 5. 378–421. 1996.

[Beyer2005] Thomas Beyer. “Concept and Implementation of Integrating the Open-Source Platform Eclipse into an Information Integration Framework for Systems Engineering (ToolNet)”. 2005.

[BizTalk2007] Microsoft. “Introducing "BizTalk Services"”. 2007.

[Brink2001] Emil Brink. “The Verse Networked 3D Graphics Platform”. 2001.

[Brown1994] Alan W. Brown, David J. Carney, Edwin J. Morris, Dennis B. Smith, and Paul F. Zarrella. “Principles of CASE tool integration”. Oxford University Press, Inc.. 1994.

[Brown1992] Alan W. Brown and John A. McDermid. “Learning From IPSE's Mistakes”. *IEEE Softw.*. IEEE Computer Society Press. 9. 23–28. 1992.

[Caselli2008] Vincenzo Caselli, Malhar Barai, and Binildas A. Christudas. “Service Oriented Architecture with Java”. Packt Publishing. 2008.

[CBESB] Bostech. “ChainBuilder ESB Reference Guide”. 2007.

[CBESBCC] Bostech. “ChainBuilder ESB Custom Component Guide”. 2007.

[Chappell2007] David Chappell. “Introducing SCA”. 2007.

[Chappell1996] David Chappell. “Understanding ActiveX and OLE: a guide for developers and managers”. Microsoft Press. 1996.

[Cheng2006] Feng Chen, Shaoyun Li, Hongji Yang, Ching-Huey Wang, and William Cheng-Chung Chu. “Feature Analysis for Service-Oriented Reengineering”. 2006.

[CoreJ2EE] Deepak Alur, John Crupi, and Dan Malks. “Core J2EE Patterns: Best Practices and Design Strategies”. Prentice Hall International. 2nd. 650. 2003.

[Chen2007] Hanwei Chen, Jianwei Yin, Lu Jin, Ying Li, and Jinxiang Dong. “JTang Synergy: A Service Oriented Architecture for Enterprise Application Integration”. *Computer Supported Cooperative Work in Design, 2007. CSCWD 2007. 11th International Conference on*. 502-507. 2007.

[Chen2007b] Chen, Jing-Ying. “Resource-Oriented Computing: Towards a Univeral Virtual Workspace”. *Advanced Information Networking and Applications Workshops, 2007, AINAW '07. 21st International Conference on*. 2. 993-1000. 2007.

[Christudas2008] Binildas A Christudas. “Service-Oriented Java Business Integration”. Packt Publishing. 2008.

[Ciurana2007] Eugene Ciurana. “Mule: A Case Study”. 2007.

[Cohen2006] Frank Cohen and Brian Bartel. “Service Governance and Virtualization For SOA”. 2006.

[Cook2007] William R. Cook. “AppleScript”. ACM. 1-1–1-21. 2007.

[Corradini2004] F. Corradini, L. Mariani, and E. Merelli. “An agent-based approach to tool integration”. 2004.

[Curbow1997] Dave Curbow and Elizabeth Dykstra-Erickson. “Designing the OpenDoc human interface”. ACM. 83–95. 1997.

[Damm2000] C. H. Damm, K. M. Hansen, M. Thomsen, and M. Tyrsted. “Tool integration: experiences and issues in using XMI and component technology”. *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*. 94–107. 2000.

[Dan2004] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. “Web services on demand: WSLA-driven automated management”. *IBM Syst. J.*. IBM Corp.. 43. 136–158. 2004.

[Davis2009] . “Open Source SOA”. Manning Publications Co.. 2009.

[Denno2003] Denno, P., Steves, M.P., Libes, D., and Barkmeyer, E.J.. “Model-driven integration using existing models”. *Software, IEEE*. 20. 59-63. 2003.

[Dmitriev2004] Sergey Dmitriev. “Language Oriented Programming: The Next Programming Paradigm”. 2004.

[Doerfel2002] Sven Dörfel and Jürgen Großmann. “Werkzeug zur Generierung von Austauschdaten in einer verteilten Umgebung”. 2002.

[DOORS] Telelogic. “Using DOORS”. 2005.

[DOORSAPI] Telelogic. “Telelogic DOORS API Manual Release 7.1”. Telelogic. 2007.

[Dubray2007] Jean-Jacques Dubray. “Composite Software Construction”. InfoQ. 2007.

[Dubray2005] Jean-Jacques Dubray. “Comparing SCA, Java EE and JBI”. 2005.

[Duftler2001] Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. “Web Services Invocation Framework (WSIF)”. 2001.

[DXL] Telelogic. “DXL Reference Manual”. Release 8.2. 2007.

[Eclipse2006] Eclipse.org. “Eclipse Platform Technical Overview”. 2006.

[Eclipse2008] . “Component Oriented Development And Assembly (CODA) with Equinox”. 2008.

[EclipseRCP] Jeff McAffer and Jean-Michel Lemieux. “Eclipse Rich Client Platform - Designing, Coding, and Packaging Java Applications”. Addison-Wesley. 1st. 552. 2005.

[EclipseSTP] Rob Cernich. “Eclipse SOA Tools Platform Project”. 2006.

[EIP] Gregor Hohpe and Bobby Woolf. “Enterprise Integration Patterns: designing, building, and deploying messaging solutions”. Addison-Wesley Professional. 1st. 686. 2003.

[Emmerich2007] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. “The impact of research on middleware technology”. *SIGSOFT Softw. Eng. Notes*. ACM. 32. 21–46. 2007.

[Erl2004] Thomas Erl. “Service-oriented architecture : a field guide to integrating XML and Web services”. Prentice Hall. 536. 2004.

[ESB] David Chappell. “Enterprise Service Bus. Theory in Practice.”. O'Reilly. 1st. 352. 2004.

[Farrell2002] Willy Farrell. “Introduction to the J2EE Connector Architecture”. 2002.

[Fielding2000] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. 2000.

[FowlerIOC] Martin Fowler. “Inversion of Control Containers and the Dependency Injection pattern”. 2004.

[Fowler1999] Martin Fowler. “Refactoring. Improving the Design of Existing Code”. Addison-Wesley. 431. 1999.

[Fremantle2002] Paul Fremantle, Sanjiva Weerawarana, and Rania Khalaf. “Enterprise services”. *Commun. ACM*. ACM Press. 45. 77–82. 2002.

[Freude2003] René Freude and Alexander Königs. “Tool integration with consistency relations and their visualisation”. 6–10. 2003.

[Fung2005] Fung, C.K., Hung, P.C.K., Linger, R.C., and Walton, G.H.. “Extending Business Process Execution Language for Web Services with Service Level Agreements Expressed in Computational Quality Attributes”. *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*. 166a-166a. 2005.

[Gautier1995] R.J. Gautier, C.W. Loftus, E.M. Sherratt, and L. Thomas. “Tool integration: experiences from the BOOST project”. *Software Engineering Environments [Conference], 1995., Proceedings*. 171-181. 1995.

[Geissler2001] Hans-Ulrich Geißler. “Automatisierte Datenkopplung von Softwarewerkzeugen am Beispiel Matlab und Doors”. 2001.

[Georgalas2005] Nektarios Georgalas and Manooch Azmoodeh. “Model Driven Integration of Standard Based OSS Components”. Eurescom Summit 2005 on Ubiquitous Services and Applications. 2005.

[Gerety1989] C Gerety. “HP Softbench: A new generation of software development tools”. 1989.

[Giampaolo1999] Dominic Giampaolo. “Practical File System Design with the Be File System”. Morgan Kaufmann Publishers. 1st. 65–67. 1999.

[GOF] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. “Design Patterns. Elements of Reusable Object-Oriented Software”. Addison-Wesley. 395. 1995.

[Goose2000] S.; Hall W.; Reich S. Goose. “Microcosm TNG: a framework for distributed open hypermedia”. *IEEE Multimedia*. 7. 52 - 60. 2000.

[Gorton2003] Ian Gorton, Dave Thurman, and Judi Thomson. “Next Generation Application Integration: Challenges and New Approaches”. IEEE Computer Society. 27. 2003.

[Greenfield2004] Jack Greenfield and Keith Short. “Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools”. John Wiley & Sons. 2004.

[Groza2007] Tudor Groza, Siegfried Handschuh, Knud Müller, Gunnar Grimnes, LeoSauermann, Enrico Minack, Mehdi Jazayeri, Cédric Mesnage, Gerald Reif, and Rósa Gudjónsdóttir. “The NEPOMUK Project - On the way to the Social Semantic Desktop”. 2007.

[Gulledge2006] Gulledge and Thomas. “What is integration?”. *Industrial Management & Data Systems*. Emerald Group Publishing Limited. 106. 5–20. 2006.

[Guo2004] Bing Guo, Yan Shen, Jun Xie, Yong Wang, and Guang-Ze Xiong. “A kind of new ToolBus model research and implementation”. *SIGSOFT Softw. Eng. Notes*. ACM. 29. 5–5. 2004.

[Haase2003] Thomas Haase. “Semi-automatic Wrapper Generation for a-posteriori Integration”. 84–88. 2003.

[Henning2006] Michael Henning. “The Rise and Fall of CORBA”. *ACM Queue Magazine*. 4. 2006.

[Hohpe2007] Gregor Hohpe. “Let's Have a Conversation”. *Internet Computing, IEEE*. 11. 78–81. 2007.

[Hohpe2006a] Gregor Hohpe. “Programming Without a Call Stack - Event-driven Architectures”. 2006.

[Hohpe2006b] Gregor Hohpe. “Workshop Report: Conversation Patterns”. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. 2006.

[Holt2006] R. Holt, A. Schürr, S. Sim, and A. Winter. “GXL: A Graph-Based Standard Exchange Format for Reengineering”. *Science of Computer Programming*. Elsevier Science Publ.. 60. 149-170. 2006.

[IEEE2006] IEEE. “IEEE Recommended Practice for CASE Tool Interconnection: Characterization of Interconnections”. *IEEE Std 1175.2-2006*. c1-36. 2007.

[JBI] Sun Microsystems. “Java Business Integration Specification 1.0 (JSR-208)”. 2005.

[JBI2] Sun Microsystems. “Java Business Integration Specification 2.0 (JSR-312)”.

[JCA15] Sun. “J2EE™ Connector Architecture Specification”. Sun Microsystems, Inc.. 2003.

[JEE5Tut] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. “The Java EE 5 Tutorial”. Sun Microsystems, Inc.. 2007.

[JMS] Richard Monson-Haefel and David Chappell. “Java Message Service”. O'Reilly & Associates, Inc.. 220. 2000.

[JNBridgePro2008] JNBridge, LLC.. “A Technical Overview of JNBridgePro”. 2008.

[Jones2005] Steve Jones. “Toward an Acceptable Definition of Service”. *IEEE Software*. 22. 2005.

[Julienne1994] Astrid M. Julienne and Brian Holtz. “ToolTalk and open protocols: inter-application communication”. Prentice-Hall, Inc.. 1994.

[Juric2007] Matjaz B. Juric, Ramesh Loganathan, Poornachandra Sarang, and Frank Jennings. “SOA Approach to Integration: XML, Web services, ESB, and BPEL in real-world SOA projects”. Packt Publishing. 2007.

[Kacmar1991] Charles J. Kacmar and John J. Leggett. “PROXHY: a process-oriented extensible hypertext architecture”. *ACM Trans. Inf. Syst.*. ACM. 9. 399–419. 1991.

[Karsai2003] Gabor Karsai, Andras Lang, and Sandeep Neema. “Tool integration patterns”. 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). 2003.

[Kaye2003] Doug Kaye. “Loosely Coupled: The Missing Pieces of Web Services”. RDS Press. 2003.

[Keller2003] Keller, Alexander and Ludwig, Heiko. “The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services”. *Journal of Network and Systems Management*. 11. 57–81. 2003.

[Kernighan1976] Brian W. Kernighan and P. J. Plauger. “Software Tools”. Addison Wesley. 1976.

[Klar2008] Felix Klar, Sebastian Rose, and Andy Schürr. “A Meta-model-Driven Tool Integration Development Process”. 201-212. 2008.

[Kramler2006] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. “Towards a semantic infrastructure supporting model-based tool integration”. ACM. 43–46. 2006.

[Lan2004] Qingguo Lan, Shufen Liu, Lu Han, and Ming Qu. “Study and realization of the inter-application communication methods”. *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*. 2. 124-127 Vol.2. 2004.

[Liang1999] Sheng Liang. “The Java™ Native Interface”. Addison-Wesley. 318. 1999.

[Linthicum1999] David S. Linthicum. “Enterprise Application Integration”. Addison-Wesley Professional. 1999.

[Liu2006] Na Liu. “Visual languages for event integration specification”. ACM. 969–972. 2006.

[Maheshwari2003] Piyush Maheshwari. “Enterprise Application Integration using a Component-based Architecture”. *COMPSAC*. 27. 2003.

[Mauritz2005] Axel Mauritz, Andreas Keis, Daniel Ratiu, and Andreas Günzler. “The Integration Framework ToolNet - Vision, Architecture and related Approaches”. 2005.

[McAfee2006] Andrew P. McAfee. “Enterprise 2.0: The Dawn of Emergent Collaboration”. *MIT Sloan Management Review*. 47. 21–28. 2006.

[Medvidovic2002] Nenad Medvidovic. “On the role of middleware in architecture-based software development”. ACM. 299–306. 2002.

[Mellor2003] Mellor, S.J., Clark, A.N., and Futagami, T.. “Model-driven development - Guest editor's introduction”. *Software, IEEE*. 20. 14-18. 2003.

[Menge2007] Falko Menge. “Enterprise Service Bus”. *FREE AND OPEN SOURCE SOFTWARE CONFERENCE 2007*. 2007.

[Meyer2001] Bertrand Meyer. “What to Compose - Going beyond the defintion of components as units of composition requires asking what and how we compose.”. *DDJ*. 2001.

[Michaels1993] K. Michaels. “Defining an architecture for control integration”. *Software Engineering Environments Conference, 1993. Proceedings*. 63-71. 1993.

[Microsoft1996] Microsoft. “OLE Automation programmer's reference: creating programmable 32-bit applications”. Microsoft Press. 1996.

[Mitschke2005] Andreas Mitschke. “Aircraft system definition with a flexible integrated tool infrastructure”. 2005.

[Mos2008] Adrian Mos, Alain Boulze, Samuel Quaireau, and Claude Meynier. “Multi-layer perspectives and spaces in SOA”. ACM. 69–74. 2008.

[Murthy2004] Sudarshan Murthy, David Maier, Lois Delcambre, and Shawn Bowers. “Putting integrated information in context: superimposing conceptual models with SPARCE”. Australian Computer Society, Inc.. 71–80. 2004.

[NascimentoS2007] Francisco Assis M. do Nascimento, Marcio F. S. Oliveira, and Flavio Rech Wagner. “ModES: Embedded Systems Design Methodology and Tools based on MDE”. *Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES '07. Fourth International Workshop on*. 67-76. 2007.

[Nepal2008] Nepal, Surya, Zic, John, and Chen, Shiping. “WSLA+: Web Service Level Agreement Language for Collaborations”. *Services Computing, 2008. SCC '08. IEEE International Conference on*. 2. 485-488. 2008.

[Neward2007] Ted Neward. “Best of Both Worlds: Java & .NET for Fun & Profit”. 2007.

[Niblett2005] P. Niblett and S. Graham. “Events and service-oriented architecture: The OASIS Web Services Notification specifications”. *IBM Systems Journal*. 44. 669–886. 2005.

[Ning2008] Fu Ning, Zhou Xingshe, Wang Kaibo, and Zhan Tao. “Distributed Enterprise Service Bus Based on JBI”. *Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08. The 3rd International Conference on*. 292-297. 2008.

[O'Reilly2005] Tim O'Reilly. “What Is Web 2.0 - Design Patterns and Business Models for the Next Generation of Software”. 2005.

[OASIS2006] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, and Rebekah Metz. “OASIS Reference Model for Service Oriented Architecture 1.0”. 2006.

[OASIS2008] Jeff A. Estefan, Ken Laskey, Francis G. McCabe, and Danny Thornton. “OASIS Reference Architecture for Service Oriented Architecture 1.0”. 2008.

[OMG] The Object Management Group. “OMG Specifications and Process: The Big Picture”. 2007.

[OMG2004] Object Management Group. “Open Tool Integration Framework”. 2004.

[OpenAdaptor2007] OpenAdaptor.org. “OpenAdaptor Whitepaper”. 2007.

[OpenESB] Sun. “Project OpenESB”.

[OpenSpan2008] Inc. OpenSpan. “OpenSpan Whitepaper”. 2008.

[OSGi2006] OSGi. “OSGi Service Platform Core Specification”. OSGi Alliance. 2006.

[OSGi2007] . “OSGi 4.1 Technical Whitepaper”. 2007.

[Ousterhout1994] John K. Ousterhout. “Tcl and the Tk Toolkit”. Addison-Wesley Professional. 1994.

[Ousterhout1990] John K. Ousterhout. “Tcl: An embeddable Command Language”. 133–146. 1990.

[Parker1992] B. Parker. “Introducing EIA-CDIF: the CASE Data Interchange Format Standard”. *Assessment of Quality Software Development Tools, 1992., Proceedings of the Second Symposium on*. 74-82. 1992.

[Parker2006] K. Parker. “Integration and Interoperability of Application Lifecycle Management Tools”. 2. 2006.

[Perry1992] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”. *Software Enginnering Notes*. 17. 13. 1992.

[Pesola2008] Pesola, J P., Eskeli, J., Parviainen, P., Kommeren, R., and Gramza, M.. “Experiences of Tool Integration: Development and Validation”. *Enterprise Interoperability III*. 499–510. 2008.

[PofEAA] Martin Fowler. “Patterns of Enterprise Application Architecture”. Addison-Wesley Professional. 1st. 560. 2002.

[Pohl1999] Klaus Pohl, Klaus Weidenhaupt, Ralf Dömges, Peter Haumer, Matthias Jarke, and Ralf Klamma. “PRIME — toward process-integrated modeling environments”. *ACM Trans. Softw. Eng. Methodol.*. ACM. 8. 343–410. 1999.

[PLUSS] Magnus Eriksson, Henrik Morast, Jürgen Börstler, and Kjell Borg. “The PLUSS Toolkit - Extending Telelogic DOORS and IBM Rational Rose to Support Product Line Use Case Modeling”. *ASE*. ACM Press. 300–304. 2005.

[POSA] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. “Pattern-Oriented Software Architecture”. Wiley. 1. 2001.

[POSA4] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. “Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing”. Wiley & Sons. 2007.

[Rademakers2008] Tijs Rademakers and Jos Dirksen. “Open Source ESBs in Action”. Manning. 2008.

[Raj2006] Gopalan Suresh Raj, Binod PG, Keith Babo, and Rick Palkovic. “Implementing Service-Oriented Architectures (SOA) with the Java EE 5 SDK”. Sun Microsystems, Inc.. 65. 2006.

[Reiss1990] S.P. Reiss. “Connecting Tools Using Message Passing in the Field Environment”. *Software, IEEE*. 7. 57–66. 1990.

[Ruiz2008] Ruiz, J.L., Duenas, J.C., and Cuadrado, F.. “A Service Component Deployment Architecture for e-Banking”. *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*. 1369-1374. 2008.

[Rymer2007] Rymer. “Web services no interop cure-all”. 2007.

[SAIP] Len Bass, Paul Clements, and Rick Kazman. “Software Architecture in Practice”. Addison-Wesley. 2nd. 512. 2003.

[Salter2008] David Salter and Frank Jennings. “Building SOA Composite Applications using NetBeans6”. Packt Publishing. 2008.

[Samuelson2006] Pamela Samuelson. “IBM's pragmatic embrace of open source”. *Commun. ACM*. ACM. 49. 21–25. 2006.

[Sauermann2008] Leo Sauermann and Sebastian Trüg. “Case Study: KDE 4.0 Semantic Desktop Search and Tagging”. 2008.

[SCA] Various. “Service Component Architecture - Building Systems using a Service Oriented Architecture”. 2005.

[Schmidt2006] Schmidt, D.C.. “Guest Editor's Introduction: Model-Driven Engineering”. *Computer*. 39. 25-31. 2006.

[Schmietendorf2004] Schmietendorf, A., Dumke, R., and Reitz, D.. “SLA management - challenges in the context of Web-service-based infrastructures”. *Web Services, 2004. Proceedings. IEEE International Conference on*. 606-613. 2004.

[SDO2007b] . “Service Data Objects White Paper”. 2007.

[SEDA2001] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”. 2001.

[Sharma2001] Rahul Sharma, Beth Stearns, and Tony Ng. “J2EE Connector Architecture and Enterprise Application Integration”. Pearson Education. 416. 2001.

[Slott2008] Jordan Slott. “Project Wonderland Software Architecture”. 2008.

[Sneed2005] Harry M. Sneed. “Wrapping Legacy Software for Reuse in a SOA”. 2005.

[Sriplakich2008] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervals. “Collaborative software engineering on large-scale models: requirements and experience in ModelBus”. ACM. 674–681. 2008.

[Sun1993] Inc. Sun Microsystems. “The ToolTalk Service: an inter-operability solution”. Sun Microsystems, Inc.. 1993.

[Sun2004] Inc. Sun Microsystems. “Java Business Integration Vision”. 2004.

[SunGlassFish] Sun. “GlassFish Community”. 2007.

[Sutherland2002] Jeff Sutherland and Willem-Jan van den Heuvel. “Enterprise Application Integration and Complex Adaptive Systems”. *Commun. ACM*. ACM Press. 45. 59–64. 2002.

[Szyperski2002] Clemens Szyperski. “Component Software”. Addison-Wesley. 2nd. 2002.

[Ten-Hove2006] Ron Ten-Hove. “Using JBI for Service-Oriented Integration (SOI)”. 11. 2006.

[Terzidis2007] Kostas Terzidis. “Algorithmic Architecture”. Architectural Press. 147. 2006.

[Thomas1992] I. Thomas and B.A. Nejmeh. “Definitions of tool integration for environments”. *Software, IEEE*. 9. 29-35. 1992.

[Touzi2007] Touzi, Jihed, Lorré, Jean-Pierre, Bénaben, Frédérick, and Pingaud, Hervé. “Interoperability through Model-based Generation: The Case of the Collaborative Information System (CIS)”. *Enterprise Interoperability*. 407–416. 2007.

[Tratt2005] Laurence Tratt. “Model transformations and tool integration”. *Software and Systems Modeling*. 4. 112–122. 2005.

[Trowbridge2004] David Trowbridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu, and E.G. Nadhan. “Integration Patterns”. Microsoft Corporation. 3. 2004.

[UMLEAI] OMG. “UML Enterprise Application Integration, V1.0”. 2004.

[VanHorn1989] VanHorn, E.C. and Rezac, R.R.. “Experience with the D-BUS Architecture for a Design Automation Framework”. *Design Automation, 1989. 26th Conference on*. 209-214. 1989.

[Verrall1992] M.S. Verrall and L. Morgan. “Tool integration in CASE environments: the Software Bus”. *Computer-Aided Software Engineering, 1992. Proceedings., Fifth International Workshop on*. 46-49. 1992.

[Vinoski2003] Steve Vinoski. “Integration with Web Services”. *IEEE Internet Computing*. 75–77. 2003.

[Vinoski2005] Steve Vinoski. “Java Business Integration”. *IEEE Internet Computing*. 9. 89-91. 2005.

[Voelter2006] Markus Völter and Thomas Stahl. “Model-Driven Software Development”. Wiley & Sons. 1st. 2006.

[Waldo1994] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. “A Note on Distributed Computing”. 1994.

[Walter2006] Torsten Walter. “Konzept und Umsetzung für das Reengineering eines Informations-Integrations-Frameworks zur System-Entwicklung (ToolNet) als Eclipse-Anwendung”. 2006.

[Warboys2005] B. Warboys, B. Snowdon, R. M. Greenwood, Wykeen Seet, I. Robertson, R. Morrison, D. Balasubramaniam, G. Kirby, and K. Mickan. “An Active-Architecture Approach to COTS Integration”. *Software, IEEE*. 22. 20–27. 2005.

[Wasserman1989] Anthony I. Wasserman. “Tool integration in software engineering environments”. Springer-Verlag New York, Inc.. 137–149. 1989.

[Welsh2002] Matthew David Welsh. “An Architecture for Highly Concurrent, Well-Conditioned Internet Services”. 2002.

[Wicks2006] M. N. Wicks. “Tool Integration within Software Engineering Environments: An Annotated Bibliography”. 2006.

[Wicks2007] M.N. Wicks and R.G. Dewar. “A new research agenda for tool integration”. *Journal of Systems and Software*. 80. 1569–1585. 2007.

[Wiil1995] U. K. Wiil. “HyperDisco: An Object-Oriented Hypermedia Framework for Flexible Software System Integration”. *COMPSAC'95*. IEEE CS Press. 19. 298-305. 1995.

[Williams1995] David Williams and Timothy O'BrienT. “Software without borders: applications that collaborate”. MIT Press. 127–156. 1995.

[Woolf2006] Bobby Woolf. “Event-Driven Architecture and Service-Oriented Architecture”. 2006.

[WS-BPEL20] OASIS. “Web service business Process Execution Language Version 2.0 Specification”. 2007.

[Yang2007] Zhihui Yang and Michael Jiang. “Using Eclipse as a Tool-Integration Platform for Software Development”. *IEEE Softw.*. IEEE Computer Society Press. 24. 87–89. 2007.

[Yap2005] N. Yap, H.C. Chiong, J. Grundy, and R. Berrigan. “Supporting dynamic software tool integration via Web service-based components”. *Software Engineering Conference, 2005. Proceedings. 2005 Australian*. 160-169. 2005.

[Young2003] Ralph R. Young. “The Requirements Engineering Handbook”. Artech House. 2003.

[Zahavi1999] Ron Zahavi. “Enterprise Application Integration with CORBA”. Wiley & Sons. 1st. 560. 1999.

[Zou2006] Zhile Zou and Zhenhua Duan. “Building Business Processes or Assembling Service Components: Reuse Services with BPEL4WS and SCA”. *Web Services, 2006. ECOWS '06. 4th European Conference on*. 138-147. 2006.

Online Resources

[ActiveMQ] Apache Software Foundation. *Apache ActiveMQ*. “Apache ActiveMQ”. http://activemq.apache.org/. 2008.

[Altman2007] Ross Altman. Sun Microsystems, Inc.. *What is a Composite Application?*. “What is a Composite Application?”. http://www.sun.com/third-party/global/layer7/collateral/r-altman-soa-discoverydays.pdf. 2007.

[Anageda] *Anageda*. “Anageda”. https://anegada.dev.java.net/. 2008.

[ApacheFelix] Apache Software Foundation. *Apache Felix*. “Apache Felix”. http://felix.apache.org/site/index.html.

[ApacheODE] Apache Software Foundation. *Apache ODE*. “Apache ODE”. http://ode.apache.org/.

[ApachePOI] Apache Software Foundation. *Apache POI - Java API To Access Microsoft Format Files*. “Apache POI - Java API To Access Microsoft Format Files”. http://poi.apache.org/.

[ApacheSynapse] *Apache Synapse Enterprise Service Bus*. “Apache Synapse Enterprise Service Bus”. http://synapse.apache.org/.

[ApacheWSIF] Apache Software Foundation. *Apache Web Services Invocation Framework*. “Apache Web Services Invocation Framework”. http://ws.apache.org/wsif/. 2006.

[Apatar] Apatar, Inc.. *Apatar Open Source Data Integration*. “Apatar Open Source Data Integration”. http://www.apatarforge.org/.

[AutoSAR] *Automotive Open System Architecture*. “Automotive Open System Architecture”. http://www.autosar.org/.

[Baer2007] Tony Baer. Computerwire. *Swiss Software Firm Introduces Executable UML*. “Swiss Software Firm Introduces Executable UML”. http://www.computerwire.com/industries/research/?pid=AFDE00B6-6F8E-4901-A9FB-F13E671BA592&type=CW%20News. 2007-04.

[Balasbanmugam2008] Prabhu Balashanmugam and Yanbing Lu. *The Role of Event-Driven Architecture in Business Applications*. “The Role of Event-Driven Architecture in Business Applications”. http://java.sys-con.com/author/6791. 2008-08.

[Burcham2005] Bill Burcham. *Baby Steps to Synergistic Web Apps*. “Baby Steps to Synergistic Web Apps”. http://lesscode.org/2005/10/21/baby-steps-to-synergistic-web-apps/. 2005-10.

[Burton2004] Ross Burton. IBM Corp.. *Connect desktop apps using D-BUS*. “Connect desktop apps using D-BUS”. http://www-128.ibm.com/developerworks/linux/library/l-dbus.html. 2004-07.

[Carr2007] Harold Carr and Arun Gupta. Sun Microsystems, Inc.. *Takes two to Tango: Java Web Services and .NET Interoperability*. “Takes two to Tango: Java Web Services and .NET Interoperability”. TS-4865 at JavaOne2007. http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-4865.pdf. 2007-05.

[ChainBuilder] Bostech Corp.. *ChainBuilder ESB*. “ChainBuilder ESB”. http://www.chainforge.net/. 2008-08.

[Ciurana2008] Eugene Ciurana. *Son of SOA: Resource-Oriented Computing and Event Driven Architectures*. “Son of SOA: Resource-Oriented Computing and Event Driven Architectures”. http://www.ciurana.eu/TSSJS2008/ROC.pdf. 2008-03.

[Codemesh2006] Codemesh, Inc.. *Codemesh Technology comparison*. “Codemesh Technology comparison”. http://codemesh.com/technology.html. 2006.

[Coalevo] Verein zur Förderung der Internetkommunikation (VFI). *The Coalevo Project*. “The Coalevo Project”. http://www.coalevo.net/.

[DaVinciVM2008] Sun Microsystems, Inc.. *The Da Vinci Machine Project*. “The Da Vinci Machine Project”. http://openjdk.java.net/projects/mlvm/. 2008.

[D-BUS] freedesktop.org. *D-BUS*. “D-BUS”. http://dbus.freedesktop.org/. 2008.

[Drools] JBoss.org. *JBoss Drools*. “JBoss Drools”. http://www.jboss.org/drools/. 2008.

[EclipseE4] Eclipse Foundation. *Eclipse E4*. “Eclipse E4”. http://wiki.eclipse.org/E4. 2008.

[EclipseLink2008] The Eclipse Foundation. *Eclipse Persistence Services Project (EclipseLink)*. “Eclipse Persistence Services Project (EclipseLink)”. http://www.eclipse.org/eclipselink/. 2008.

[EclipseRAP] *Eclipse Rich Ajax Platform*. “Eclipse Rich Ajax Platform”. http://www.eclipse.org/rap. 2008.

[EclipseSCA] Eclipse Foundation. *The Eclipse STP/SCA Subproject*. “The Eclipse STP/SCA Subproject”. http://www.eclipse.org/stp/sca/.

[eRCP] *Eclipse Embedded Rich Client Platform*. “Eclipse Embedded Rich Client Platform”. http://www.eclipse.org/ercp/. 2008.

[Esper] Espertech. *Event Stream Intelligence: Esper & NEsper*. “Event Stream Intelligence: Esper & NEsper”. http://esper.codehaus.org/.

[Fabric3] Codehaus.org. *Fabric3*. “Fabric3”. http://fabric3.codehaus.org/.

[Flickr] *Flickr*. “Flickr”. http://www.flickr.com/.

[Fowler2005] Martin Fowler. http://www.martinfowler.com/articles/languageWorkbench.html. 2005-06.

[FUSE] Progress. *FUSE Open Source Community*. “FUSE Open Source Community”. http://fusesource.com/. 2008.

[GASwerk] *GASwerk - Geronimo Application Server Assemblies*. “GASwerk - Geronimo Application Server Assemblies”. http://gaswerk.sourceforge.net/.

[Gigaspaces] Gigaspaces Technologies. *GigaSpaces eXtreme Application Platform (XAP)*. “GigaSpaces eXtreme Application Platform (XAP)”. http://www.gigaspaces.com/.

[Glassbox] *Glassbox*. “Glassbox”. http://www.glassbox.com/glassbox/Home.html. 2008.

[GoogleMaps] *Google Maps*. “Google Maps”. http://maps.google.com/.

[GoogleME] Google, Inc.. *Google Mashup Editor*. “Google Mashup Editor”. http://code.google.com/gme/docs/gettingstarted.html.

[GridGain] GridGain. *GridGain Grid Computing*. “GridGain Grid Computing”. http://www.gridgain.com/.

[Grigonis2008] Richard Grigonis. IP Communications Group. *IBM Secures Web 2.0 Mashups with SMash*. “IBM Secures Web 2.0 Mashups with SMash”. http://opensourcepbx.tmcnet.com/topics/development-tools/articles/22834-ibm-secures-web-20-mashups-with-smash.htm. 2008-03.

[Hinchcliffe2006] Dion Hinchcliffe. *Making the Most of the Web: Creating Great Mashups*. “Making the Most of the Web: Creating Great Mashups”. http://web2.socialcomputingmagazine.com/making_the_most_of_the_web_creating_great_mashups.htm. 2006-05.

[IFL2008] Mark Saunders. Sun Microsystems, Inc.. *Integration Flow Language Overview*. “Integration Flow Language Overview”. http://wiki.open-esb.java.net/Wiki.jsp?page=IntegrationFlowLanguageOverview. 2008-10.

[Jahn2008] Mirko Jahn. *Some thought on the OSGi R4.2 early draft*. “Some thought on the OSGi R4.2 early draft”. http://osgi.mjahn.net/2008/08/28/some-thought-on-the-osgi-r42-early-draft/. 2008-08.

[JavaRules2008] Various. *The Java Business Rules Community*. “The Java Business Rules Community”. http://www.javarules.org/. 2008.

[JBIDev] Sun. *Developing JBI Components*. “Developing JBI Components”. https://open-esb.dev.java.net/public/jbi-comp-examples/Developing_JBI_Components.html.

[JBossJBPM] JBoss.org. *JBoss jBPM*. “JBoss jBPM”. http://www.jboss.org/jbossjbpm/. 2007.

[Jitterbit] Jitterbit. *Jitterbit Enterprise Integration*. “Jitterbit Enterprise Integration”. http://www.jitterbit.com/Product/enterprise-integration.

[JNA] Sun Microsystems, Inc.. *Java Native Access (JNA): Pure Java Access to native libraries*. “Java Native Access (JNA): Pure Java Access to native libraries”. https://jna.dev.java.net/.

[JPF] *Java Plugin Framework*. “Java Plugin Framework”. http://jpf.sourceforge.net/.

[JRuby] Codehaus Foundation. *JRuby - Java powered Ruby implementation*. “JRuby - Java powered Ruby implementation”. http://jruby.codehaus.org/. 2006.

[JSR94] Daniel Selman. *JSR 94: Java Rule Engine API*. “JSR 94: Java Rule Engine API”. http://jcp.org/en/jsr/detail?id=94. 2004-08.

[JSR223] Sun Microsystems, Inc.. *JSR 223: Scripting for the Java Platform*. “JSR 223: Scripting for the Java Platform”. http://jcp.org/en/jsr/detail?id=223.

[JSR292] Danny Coward. Sun Microsystems, Inc.. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform*. “JSR 292: Supporting Dynamically Typed Languages on the Java Platform”. http://jcp.org/en/jsr/detail?id=292. 2008-05.

[Kieviet2007] Frank Kieviet, Alex Fung, Sherry Weng, and Srinivasan Chikkala. Sun Microsystems, Inc.. *Developing Components for Java Business Integration: Binding Components and Service Engines*. “Developing Components for Java Business Integration: Binding Components and Service Engines”. http://mediacast.sun.com/users/Frank.Kieviet/media/JavaOne07-BOF8847-JBIComponents.pdf. 2007.

[Kinnumpurath2005] Meeraj Kinnumpurath. *JBI - A Standard-Based Approach for SOA in Java*. “JBI - A Standard-Based Approach for SOA in Java”. http://www.theserverside.com/tt/articles/article.tss?l=JBIforSOA. 2005-12.

[Knopflerfish] The Knopflerfish Project. *Knopflerfish Open Source OSGi*. “Knopflerfish Open Source OSGi”. http://www.knopflerfish.org/.

[KROSS] KDE Community. *The KROSS Vision*. “The KROSS Vision”. http://kross.dipe.org/vision.html.

[Lachor2008] Kris Lachor. *Systems Integration with Openadaptor*. “Systems Integration with Openadaptor”. http://java.sys-con.com/node/535350. 2008-10.

[LDTP] *Linux Desktop (GUI Application) Testing Project (LDTP)*. “Linux Desktop (GUI Application) Testing Project (LDTP)”. http://ldtp.freedesktop.org/wiki/.

[Mashable] *Mashable - All that's New on the Web*. “Mashable - All that's New on the Web”. http://www.mashable.com/.

[MC4J] *MC4J Management Console*. “MC4J Management Console”. http://mc4j.org/.

[MicrosoftPopfly] Microsoft Corp.. *Popfly*. “Popfly”. http://www.popfly.com/.

[Mule] MuleSource Inc.. *Mule 2.0 Getting Started Guide*. “Mule 2.0 Getting Started Guide”. http://www.mulesource.org/display/MULE2INTRO/Home. 2008.

[OpenAjax] OpenAjax. OpenAjax Alliance. *Next-Generation Applications Using Ajax and OpenAjax*. “Next-Generation Applications Using Ajax and OpenAjax”. http://www.openajax.org/whitepapers/Next-Generation%20Applications%20Using%20Ajax%20and%20OpenAjax.php#Mashups.2C_dashboards_and_other_composite_applications.

[OpenArchitectureWare] Eclipse. *OpenArchitectureWare*. “OpenArchitectureWare”. http://www.eclipse.org/gmt/oaw/.

[OpenCSA2008] OASIS. *OASIS Open Composite Services Architecture (CSA) Member Section*. “OASIS Open Composite Services Architecture (CSA) Member Section”. http://www.oasis-opencsa.org/. 2008.

[OpenDMK] Sun Microsystems, Inc.. *Project OpenDMK*. “Project OpenDMK”. https://opendmk.dev.java.net/. 2007.

[OpenESB2008] Derek Frankforth. *REST support in OpenESB*. “REST support in OpenESB”. http://wiki.open-esb.java.net/Wiki.jsp?page=RESTSupport. 2008-09.

[OpenESBScriptingSE] Sun Microsystems, Inc.. *OpenESB Scripting SE*. “OpenESB Scripting SE”. http://wiki.open-esb.java.net/Wiki.jsp?page=ScriptingSE. 2008-06.

[OpenID] OpenID Foundation. *OpenID*. “OpenID”. http://openid.net/.

[OpenSearch] A9.com, Inc.. *OpenSearch*. “OpenSearch”. http://www.opensearch.org/Home. 2008.

[OpenSocial] Google, Inc.. *OpenSocial*. “OpenSocial”. http://code.google.com/apis/opensocial/. 2008.

[OSOA2007] *Relationship of SCA and JBI*. “Relationship of SCA and JBI”. http://www.osoa.org/display/Main/Relationship+of+SCA+and+JBI. 2007.

[Petals] *PEtALS Open Source ESBs*. “PEtALS Open Source ESBs”. http://petals.objectweb.org/.

[pjb] *php/Java bridge*. “php/Java bridge”. http://php-java-bridge.sourceforge.net/pjb/.

[ProgrammableWeb] *ProgrammableWeb - Mashups, APIs, and the Web as Platform*. “ProgrammableWeb - Mashups, APIs, and the Web as Platform”. http://www.programmableweb.com/.

[ProjectFuji] Sun Microsystems, Inc.. *Project Fuji*. “Project Fuji”. https://fuji.dev.java.net/.

[Quercus] Caucho. *Quercus PHP 5 Java port*. “Quercus PHP 5 Java port”. http://quercus.caucho.com/.

[Raj2007] Gopalan Suresh Raj. *How to Deliver Composite Applications with Java, WS-BPEL & SOA*. “How to Deliver Composite Applications with Java, WS-BPEL & SOA”. http://java.sys-con.com/node/358049. 2007-04.

[RIFWG2008] W3C. *RIF Working Group*. “RIF Working Group”. http://www.w3.org/2005/rules/wiki/RIF_Working_Group.

[SCAUML] OpenSOA. *SCA Expressed as a UML Model*. “SCA Expressed as a UML Model”. http://www.osoa.org/display/Main/SCA+Expressed+as+a+UML+Model.

[SCOrWare] *SCOrWare*. “SCOrWare”. http://www.scorware.org/projects/en.

[SDO2007a] Graham Barber. OpenSOA. *Service Data Objects Home*. “Service Data Objects Home”. http://www.osoa.org/display/Main/Service+Data+Objects+Home. 2007-11.

[ServiceMix] Apache Software Foundation. *Apache ServiceMix, the Agile Open Source ESB*. “Apache ServiceMix, the Agile Open Source ESB”. http://servicemix.apache.org/. 2008.

[ServiceMixScript2008] Lars Heinemann. Apache Software Foundation. *Servicemix Scripting Component*. “Servicemix Scripting Component”. http://servicemix.apache.org/servicemix-scripting.html.

[Snyder2007] Bruce Snyder. IONA Technologies. *Service Oriented Integration With Apache ServiceMix*. “Service Oriented Integration With Apache ServiceMix”. http://servicemix.apache.org/articles.data/SOIWithSMX.pdf. 2007.

[Sommers2005] Frank Sommers. *Service-Oriented Java Business Integration*. “Service-Oriented Java Business Integration”. http://www.artima.com/lejava/articles/jbi.html. 2005-08.

[Spagic] Engineering. *Spagic SOA Enterprise Integration Platform*. “Spagic SOA Enterprise Integration Platform”. http://www.spagic.org/ecm/faces/public/guest/home/solutions/spagic.

[Spring] SpringSource. *Spring*. “Spring”. http://www.springframework.org/.

[SpringDM] SpringSource. *Spring Dynamic Modules for OSGi*. “Spring Dynamic Modules for OSGi”. http://www.springframework.org/osgi.

[SpringIntegration] SpringSource. *Spring Integration*. “Spring Integration”. http://www.springframework.org/spring-integration.

[SpringRepository] SpringSource. *SpringSource Launches Enterprise Bundle Repository for OSGi*. “SpringSource Launches Enterprise Bundle Repository for OSGi”. http://www.springsource.com/node/734. 2008.

[Sun2006] Sun Microsystems, Inc.. *Java EE Service Engine Overview*. “Java EE Service Engine Overview”. http://download.java.net/general/open-esb/docs/jbi-components/jee-se.html. 2006.

[Swordfish] Eclipse Foundation. *Swordfish SOA Runtime Framework Project*. “Swordfish SOA Runtime Framework Project”. http://www.eclipse.org/swordfish/. 2008.

[Tuscany2008] Apache Software Foundation. *Apache Tuscany*. “Apache Tuscany”. http://tuscany.apache.org/. 2008-05.

[Verse] Quelsolaar. *Verse*. “Verse”. http://www.quelsolaar.com/verse/.

[Walker2007] Peter Walker. Sun Microsystems, Inc.. *What's coming with JBI 2.0*. “What's coming with JBI 2.0”. presentation. http://jazoon.com/download/presentations/1841.pdf. 2007-06.

[WS-BPEL2007] IBM Corp.. *WS-BPEL Extension for People (BPEL4People), Version 1.0*. “WS-BPEL Extension for People (BPEL4People), Version 1.0”. http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/. 2007-06.

[WSI] *Web Services Interoperability Organization*. “Web Services Interoperability Organization”. http://www.ws-i.org/.

[WSIT] Sun Microsystems, Inc.. *WSIT: Project Tango*. “WSIT: Project Tango”. https://wsit.dev.java.net/. 2008.

[WSPER2007] Jean-Jacques Dubray. WSPER.org. *WSPER: An abstract SOA framework*. “WSPER: An abstract SOA framework”. http://www.wsper.org/primer.html. 2007-08.

[XAware] XAware. XAware.org. *XAware Open Source Data Integration*. “XAware Open Source Data Integration”. http://www.xaware.org/.

[Xcalia] Xcalia. Xcalia S.A.. *Xcalia Dynamic Data Integration Software*. “Xcalia Dynamic Data Integration Software”. http://www.xcalia.com/.

[XPCOM] The Mozilla Foundation. *Mozilla Cross Platform Component Object Model developer site*. “Mozilla Cross Platform Component Object Model developer site”. http://developer.mozilla.org/en/XPCOM.

[YahooPipes] Yahoo, Inc.. *Yahoo Pipes*. “Yahoo Pipes”. http://pipes.yahoo.com/pipes/.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Colophon

In the spirit of this work, this work was almost entirely produced using open source or free tools and technology[1], using Kubuntu Linux 8.04 as operating system.

The source documents were written in DocBook5 XML using XML Mind's free Java-based XML editor (XXE 4.1.0). Apache FOP (0.95) was used to render the output as PDF, using the XSL stylesheets (1.74) from docbook.org with custom extensions.

Illustrations were done using Inkscape (0.46) and OpenOffice Draw (3). The Zim desktop wiki (0.26-2) proved to be a valuable tool for organizing or at least categorizing and connecting my thoughts.

For managing references, the open source Java tool JabRef 2.4 was used, together with bib2db5 for formatting online references.

[1] with the notable exception of Microsoft Visio2003, which was necessary to work with the EnterpriseIntegrationPatterns-template kindly provided by Gregor Hohpe.