DISSERTATION

Efficient Space-Based Application Development with Declarative and Aspect-Oriented Programming

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
E185
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. (FH) Fabian Schmied
0327182
Krichbaumgasse 10/3, 1120 Wien

Wien, am 15. Februar 2007

Kurzfassung

Verteilte Computeranwendungen ermöglichen es heutzutage, mit Partnern, die sich an unterschiedlichen Orten, in unterschiedlichen Ländern und sogar auf unterschiedlichen Kontinenten aufhalten, zusammenzuarbeiten und Geschäfte zu machen. Derartige Computerprogramme haben spezielle Anforderungen, wie beispielsweise die Organisation der im System eingebundenen Geräte, Abstraktion deren Unterschiede, Koordination der involvierten Prozesse und vieles mehr. Dies unterscheidet ihre Entwicklung von der lokaler Computeranwendungen, und trotz existierender Software-Engineering-Ansätze ist es immer noch eine große Herausforderung, die Budgetpläne und Entwicklungszeiten verteilter Softwareprojekte einzuhalten.

Um dies zu verbessern, muss die Effizienz der Entwicklung verteilter Anwendungen gesteigert werden, indem die Komplexität des Entwicklungsprozesses reduziert wird. Einer der wichtigsten Mechanismen zur Bewältigung von Komplexität ist Abstraktion, und so benutzt man Middleware-Systeme, um die Details verteilter Systeme zu abstrahieren. Ein besonders bemerkenswertes Middleware-Konzept ist das Modell des Space-Based Computing, das eine natürliche datenbezogene Abstraktion des zugrunde liegenden verteilten Systems bietet. Aktuelle Implementierungen dieses Modells bieten Programmierschnittstellen auf dem neuesten Stand der Technik, die es Entwicklern erlauben, die Kommunikation und Koordination verteilter Prozesse auf objektorientierte Weise über einen virtuellen Datenraum abzuwickeln. Allerdings wirken genau diese Schnittstellen dem eigentlichen Ziel der Reduktion von Komplexität durch Abstraktion entgegen, da sie die Entwickler zwingen, ihre verteilten Ziele und Absichten auf imperative Anweisungen an das Middleware-System abzubilden.

Diese Dissertation präsentiert eine neuartige Methode zur deklarativen Formulierung der Verteilungsaspekte Space-basierter Anwendungen und macht somit die explizite Formulierung imperativer Algorithmen zu diesem Zweck unnötig. Durch den Einsatz des deklarativen Mechanismus zur Quelltext-Attributierung, den die moderne objektorientierte Anwendungsplattform .NET bietet, erlaubt sie das direkte Angeben von Verteilungszielen im Quelltext; die Aufgabe des Interpretierens und Erreichens dieser Ziele wird einer entkoppelten Umgebung überlassen. Dies ermöglicht es Entwicklern, besser dokumentierte und einfacher verständliche Space-basierte verteilte Software in kürzerer Zeit und mit besserer Codequalität zu erstellen, als dies bisher möglich war.

Um dies zu realisieren greift die Arbeit Konzepte der aspektorientierten Programmierung auf. Sie betrachtet die Anforderungen der Verteilung als crosscutting Concerns and benutzt aspektorientierte Konzepte wie Aspekte, Join-

Points und Pointcuts, um das Laufzeitverhalten, das den deklarativen Zielbeschreibungen zugrunde liegt, zu implementieren. Sie präsentiert eine einfach einsetzbare und leichtgewichtige aspektorientierte Programmierumgebung für .NET, die es erlaubt, die Anforderungen Space-basierter Anwendungen sauber als modulare, wiederverwendbare Einheiten zu kapseln und in unterschiedlichsten Szenarien einzusetzen.

Auf Basis des deklarativen Modells und des aspektorientierten Rahmenwerks definiert sie einen Katalog häufig im Bereich von Space-Based Computing auftretender Anforderungen und stellt aspektorientierte Implementierungen davon in Form einer deklarativ anwendbaren Bibliothek zur Verfügung. Durch Analyse und Auswertung der Implementierungen und durch Vergleich mit traditionellen Lösungen dieser Anforderungen kann gezeigt werden, dass dieser neuartige Ansatz zu Space-Based Computing viele Vorteile bringt; er führt zu saubereren und besser gekapselten Lösungen, reduziertem Entwicklungsaufwand und qualitativ höherwertigem Quelltext.

Abstract

In the modern electronic world, distributed computer applications allow people to perform collaborative work and conduct business transactions with partners residing in different places, countries, and even continents. Such applications have distinguishing requirements, including the organization of the devices involved in the system, abstraction of their differences, coordination of the processes involved, and many more. This makes their development different from that of local applications, and despite existing software engineering and development approaches, it is still a big challenge to keep planned budget and development time frames when carrying out a distributed software project.

In order to improve on that, distributed application development must be made more efficient by reducing the complexity of the development process. One of the most important ways of tackling complexity is abstraction, and so middleware layers are used in order to abstract the details of distributed systems. A particularly remarkable concept for middleware layers is the model of space-based computing, which provides a natural data-centered abstraction of the underlying distributed system. Current implementations of this model offer state-of-the-art application programming interfaces, allowing developers to use a virtual data space for communication and coordination of distributed processes in an object-oriented way. However, exactly these interfaces effectively counteract the original goal of reducing complexity through abstraction, because they force developers to map their distributional goals and intents to imperative instructions to the middleware.

This thesis proposes a novel way of declaratively stating the distributional intents of space-based applications, replacing the need to explicitly formulate imperative algorithms. By employing the declarative source code attribution mechanism offered by the modern object-oriented .NET application platform, it enables programmers to directly specify their goals in the source code, leaving the tasks of interpreting and finding a way to achieve them to a decoupled environment. This allows developers to create better documented and more easily understandable space-based distributed software in less time and with better code quality features than before.

To make this possible, the thesis adopts notions from aspect-oriented programming. It regards the distributional intentions as cross-cutting concerns and employs the aspect-oriented concepts of aspects, join points, and pointcuts in order to implement the behavior backing the declarative goal specifications. It presents an easily adoptable light-weight aspect-oriented programming environment for .NET, which effectively allows to cleanly encapsulate and modularize

the requirements of space-based applications and to reuse them in many application scenarios.

Based on the declarative model and the aspect-oriented framework, it defines a catalog of concerns common to space-based computing and gives aspect-oriented implementations of them in the form of a declaratively applicable distributed concern library. By means of analyzing and evaluating the implementations and comparing them to traditional ways of solving these concerns, it can be shown that this novel approach to space-based computing is highly advantageous; it leads to cleaner and more encapsulated solutions, reduced effort in development, and higher quality source code.

Acknowledgements

First of all, I have to thank my supervisor, eva Kühn, who accepted me as a PhD student even though she didn't know me as I came from a different university, who successfully encouraged me to write my first publication on space-based computing and aspect-orientation, and who was always there to talk (or make an appointment for a talk) when needed. I also want to thank my second professor, Renate Motschnig, who agreed to supervise me, taking the time although I know she had many other duties.

Then, I certainly wouldn't have had the patience and motivation to go through the last three and a half years had I not been part of the space-based computing team at my institute. Martin Murth, who came to this university together with me, attended the same lectures as I did, and started working on space-based computing together with me, is a great colleague, a person to have fun with, and a friend always there to discuss issues of technical, personal, etymological, or philosophical nature. Johannes Riemer, who joined us shortly after we had moved into our office at the institute, also is a great colleague; we had many enjoyable technical and non-technical discussions. And the SBC group wouldn't be complete without Marcus Mor and Richard Mordinyi, leaders of the XVSM MozartSpaces group, who were extremely helpful and always willing to discuss aspects of space-based computing.

Of the other people at university, I especially want to thank Jens Knoop, head of our institute, for always being there when needed. He made sure that we had a workplace and a good infrastructure, and he helped with financing my attending conferences more than once. And most sincere thanks to Ulrich Neumerkel, who hired me as a tutor for logic-oriented programming and thus not only caused a steady (though small) income for me, but also made me have a lot of fun in our numerous tutor meetings, no matter whether we discussed the lectures, watched DVD trilogies, or ate pork knuckle.

And finally, I want to thank my parents, Brigitte and Christian Schmied, who supported me both mentally and financially during my pre- and postgraduate studies, making it possible for me to do exactly what I wanted. What more could one possibly desire?

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

> *«[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.»* Edsger Dijkstra, "The Humble Programmer"

Software crisis is a term which emerged at the end of the 1960s. For the first time, software costs were higher than those of the hardware; an unanticipated development, which resulted in the first big software project failures. In succession, a large number of software projects had to be canceled, many could only be completed with reduced functionality or exceeded time and budget. To remedy this situation, the concept of *software engineering* [Som01] was introduced—software development with applied principles of engineering was meant to end the crisis.

Nowadays, the term "software crisis" isn't used widely any more, but modern software development still can hardly be called effective. Starting in 1994, *The Standish Group*, a research and consulting facility specializing on IT project success rates, has published annual "chaos reports" after surveying a number of software projects. In its last publicly available report of the 3rd quarter of 2004 [Gro04], only 29% of the surveyed projects were successfully finished within the preestimated limits of cost and time. The remaining 71% were either considered challenged—finished only with cutbacks in functionality or exceeded budgets and time frames—or failed at all, as illustrated in figure 1.1; an unsatisfactory situation.

In 1986, Frederick P. Brooks, Jr., predicted in his article "No Silver Bullet" [FPB87] that no *silver bullet*, no easy cure for the software crisis would be found in the following years. Now, 20 years later, the disappointing success rate of software projects proves him right. Brooks accredited the software crisis' symptoms to two different kinds of complexity: *essential* difficulties, i.e. complexity which is inherent to the problem itself and which cannot be removed, and *accidental* difficulties, i.e. complexity imposed on the programmer because of the wrong or insufficient tools and paradigms.

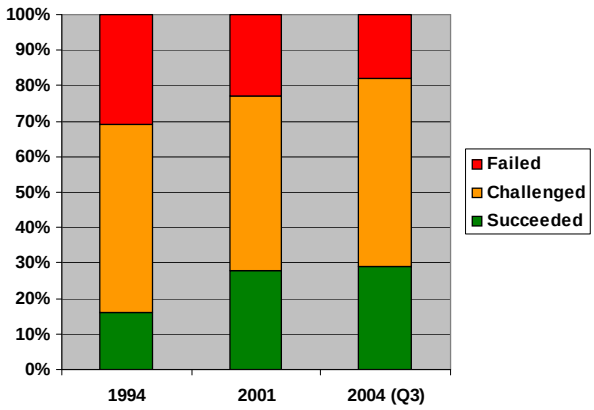

Figure 1.1: Success rate with software projects, from: *2004 Third Quarter Chaos Research Report*, The Standish Group, 2004

By definition, essential complexity must be coped with. This leaves accidental complexity as the only way to effective software development and improved project success rates, and indeed this is the topic of this thesis: to give the programmer better, more suitable tools for his tasks, reduce accidental complexity, and allow for more effective software development.

1.1 Distributed Application Development

> *A distributed computing system (DCS) interconnects many autonomous computers to satisfy the information processing needs of modern enterprises. [...] Advances in coputer and communication technologies have made possible DCSs of different sizes, shapes and forms.*
>
> Amjad Umar, "Distributed Computing"

> *You know you have one when the crash of a computer you've never heard of stops you from getting any work done* Leslie Lamport (supposedly)

This thesis concentrates on a special kind of software application development: creation of distributed applications [TS01]. Distribution is a most important issue in our modern electronic world: large information networks such as the *World Wide Web* span the globe, technologies allowing business to be conducted all over the world are available, and buzzwords like *eBusiness*, *eCollaboration*, and *Web 2.0* are setting the trend. With distributed users, distributed software applications, and distributed hardware and devices, distribution is omnipresent.

Unfortunate to software engineering, the property of distribution is inherently connected to complexity: distributed applications face requirements which automatically raise their essential complexity, examples of which include the following [TS01][CD00][eKN98]:

Organization and communication: Different devices involved in the network need to be organized, and physical network connections and network software components (such as routers) must be interfaced in order to make communication between the computers possible. Common protocols need to be devised, implemented, and accessed from the software components.

Heterogeneity: The devices in the network may vary in capabilities, hardware configuration, operating system, and installed software components. To combine them into a uniform system, these differences need to be taken into account. Even if the same application logic is to be implemented on different devices, code written for one platform might not be compatible with another one, unless some kind of abstraction mechanism is used.

Coordination: The software components making up the distributed application need to be coordinated. For example, if two components simultaneously access and modify the same data, a consistent state must be reached. If one component needs the results of another one in order to perform a task, they must be synchronized.

Security: For sensitive data, authentication and identification must be used to ensure confidentiality of the information and to guarantee that only authorized parties can modify it. In addition, the software must make sure that data consistency and integrity is maintained and that eavesdropping is prevented, for example by using strong encryption mechanisms.

Reliability: When a component breaks, no matter if it is hardware or software, the system itself and the application it is running should be able to work around the fault and keep running if possible. Reliability is achieved by making hardware more robust and providing backup computers, but it is also a property of distributed software.

Scalability: If the number of users of a distributed application gets higher than previously anticipated, the application must be able to adapt to this situation automatically—provided the hardware is capable of supporting the higher number of users. Requiring additional hardware as the number of clients rises is costly, so the software should ensure existing hardware is used as efficiently as possible, balancing the data processing load as well as possible between the computers involved in the system.

Issues such as those imply raised essential complexity, which in turn leads to a higher chance of failure for distributed application development. Therefore, it is especially important to provide the developer team of a distributed application with the right tools and mechanisms, keeping accidental complexity at a minimum.

1.2 Middleware and Network Abstractions

> *«The history has been a continuous evolution towards more and more flexiblility of techniques and tools. My message is now that this request or change has reached unprecedented levels of speed and we are moving towards what I call the open world that changes in unpredictable and unanticipated ways.»* Carlo Ghezzi, at ETAPS'06

The first and most important step to minimizing accidental complexity for distributed applications is to provide an abstraction for the distributed system and underlying network. Middleware layers [CD00] and similar network abstraction technologies do this by providing a high-level networking model, hiding the details of communication and heterogeneity. They provide services for coordination and security, and thus offer the programmer a convenient way of dealing with the essential complexity. By providing predefined networking protocols, they can—to a certain extent—remove the burdens of reliability and scalability issues from the application developer.

Good abstractions already exist. *Space-based computing* peer-to-peer technologies such as CORSO (Coordinated Shared Objects) [eK94] and XVSM (Extensible Virtual Shared Memory) [eKBM05][eKRJ05] combine the high-level networking model of a distributed shared memory (the *space* [Gel85]) with sophisticated replication protocols, which provide reliability and scalability "out of the box". The space abstraction, which is further detailed in chapter 2, allows a natural and data-centered way of communication, and its aforementioned implementations provide high-level transaction and notification services for coordination.

These tools provided, the goal of this thesis is not to reinvent the wheel by defining or discussing models of network abstraction. Instead, because of its well-defined positive properties (detailed in chaper 2), the space-based computing model and its CORSO and XVSM implementations are selected as a base for the work, going one step further and enabling programmers to make most efficient and most effective use of these existing tools. Because XVSM is currently being devised, most concrete implementations we provide will be based on CORSO's application programming interfaces (APIs) and the services it provides. Many of the results will however not be restricted to space-based computing, and some will be applicable for non-distributed applications as well.

1.2.1 Specifying Intent, Not Implementation

> *«What does a high-level language accomplish? It frees a program from much of its accidental complexity.»*
>
> Frederick P. Brooks, Jr., "No Silver Bullet"

When designing a distributed application, the services provided by the network abstraction layer have to be interfaced by the application. For example, space-based application components typically need to access objects in the shared space, add new space objects, subscribe to events which notify of changes performed by other components, and so on [eKN98]. For this, CORSO provides

language bindings: programming interfaces for general-purpose programming languages, which enable an application component to communicate with the middleware layer. Figure 1.2 illustrates this concept: application code written in Java, C++, or any .NET programming language (these bindings are currently supported by CORSO) address middleware services by invoking functionality of the language bindings. The language bindings in turn directly communicate with the CORSO kernel. In the figure, arrows represent direct and explicit communication; for example, an application component programmer must explicitly invoke the features of the language binding.

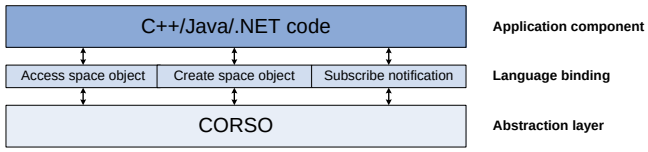

Figure 1.2: Access to the CORSO middleware using language bindings

However, is explicit communication with the middleware layer always a desirable way of implementing a distributed application? We believe not. In fact, we state that an explicit interface to the networking technology actually raises accidental complexity of an application, because it does not provide an optimal level of abstraction to the programmer. We believe that the process of formulating concerns of distribution in an imperative way, as it is necessary with explicit imperative procedural or object-oriented middleware language bindings, is not a natural way of expressing these concerns.

Instead, programmers should be provided a high-level language allowing them to express their intentions in a declarative way rather than having to specify the exact implementation steps. As early as 1983, Barbara Liskov and Robert Scheifler demonstrated linguistic support for distributed programming in a special-purpose programming language called *Argus* [LS83] [Lis88]. Other approaches followed, including the *D* language framework [LK97] developed by Cristina Lopes and Gregor Kiczales at the Palo Alto Research Center. Figure 1.3 illustrates how a special-purpose programming language could be combined with CORSO: the component is developed in the special-purpose programming language; because the programming language provides declarative features for distributional concerns, no direct and explicit communication is necessary between component code and middleware (no arrows are shown), resulting in more decoupled and more abstracted code.

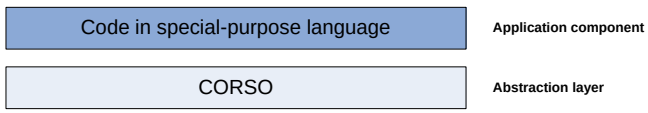

Figure 1.3: Access to the CORSO middleware using a special-purpose language

However, although both *Argus* and *D* provide high-level language features for distributed programming and thus fulfill the requirement of lowering accidental complexity of distributed software projects, they haven't been commercially successful, a fate shared of many academic programming languages. We believe

that a successful approach should provide similar high-level declarative access to middleware services from within the context of already commercially successful *general-purpose* programming languages such as $C^{\#}$ or Java.

In this thesis, we therefore propose a new approach for developing space-based distributed application components which employs the declarative facilities offered by modern object-oriented programming languages. *Source code annotations* [GJSB05], or *custom attributes* [ECM05a], as they are provided by Java and .NET-based programming languages, enable developers to declaratively specify concerns from within the context of an object-oriented program in an orthogonal way. With the help of an additional runtime environment component—which interprets the annotations and informs the middleware about the desired functionality—, these annotations can be used to indirectly interface the middleware. Figure 1.4 shows how an application component can be developed in a decoupled and abstracted way—similar to the special-purpose language-based implementation in figure 1.3—, although it is written in a general-purpose programming language. An *annotation interpreter* is used to translate the declarative concerns to explicit interaction with the middleware via language bindings, as in figure 1.2. Note that a modern programming language supporting source-code annotations must be used to implement the application component: because standard C++ does not have such a concept, an extended version such as managed C++ (MC++) or C++/CLI [ECM05c] must be used.

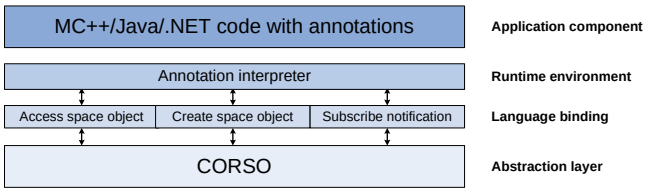

Figure 1.4: Access to the CORSO middleware using annotations

So, what are the actual benefits provided to programmers via the ability of declarative specification of concerns? Declarativity enables them to specify *intent* instead of *implementation*. It enables them to specify *goals* instead of *algorithms*. This provides a better level of abstraction, reduces accidental complexity, and, as will be shown in this thesis, significantly improves the development process of space-based distributed applications. Goal-direction aids in separation of concerns, keeping the code for single concerns together in single units of encapsulation and reducing dependencies and coupling between them. The code for each concern therefore becomes more compact, more self-contained, and more concise, making the whole application source code more readable, better understandable, more extensible, and thus better maintainable. Altogether, this considerably improves the programming process, and makes space-based application development much more efficient.

1.2.2 Aspects Make the System Recognize the Goal

Unfortunately, just writing annotations in front of a program element in source code is not enough to make an application behave in a distributed way. While

the broad intent of an attribute might be clear to the human reader, the computer has to actually perform an algorithm when the method is executed. The system running the program must therefore be made to recognize the intentional attributes used in an application and execute according behavior; in figure 1.4, the "annotation interpreter" was included for this purpose.

Currently, object-oriented programming languages, even when providing the concept of declarative source code annotations, do not offer powerful built-in mechanisms for interpretation of annotations and encapsulation of the algorithms associated with them. To remedy this, we make use of a general concept for introducing modularized additional behavior to programs: *aspect-oriented programming* [KL+97].

With aspect-oriented programming, developers divide their software's functionality into two kinds of concerns: *functional concerns* deal with what we call the application's *core* or *business logic*, i.e. the functionality implementing the main purpose of the program. *Cross-cutting concerns* deal with orthogonal functionality, such as transaction safety, data transfer to the space, error handling, or security issues. While functional concerns are implemented using object-oriented mechanisms like objects, inheritance, and aggregation, AOP provides new mechanisms for cross-cutting concerns: aspects, join points, and pointcuts. Aspect-oriented programming is further detailed in chapter 2.

This thesis adapts the aspect-oriented notions: it implements the distributional concerns as aspects tied to declarative code annotations. The aspects translate the intents specified by the programmer to algorithms executed at runtime, enabling goal-directed code to be written although algorithms are executed.

1.3 Distributed Whiteboard Example

To illustrate the benefits brought by an annotation-based and aspect-oriented middleware interface and to quickly demonstrate the concepts we present, we will use a collaborative example application, for which the space-based paradigm is especially well-suited [eKS05a]: a distributed whiteboard. Users should be able to login to this application from different places and collaboratively create drawings consisting of graphical shapes. Each user sees the modifications performed by other participants and can join in the drawing process, adding, removing, and manipulating shapes. In order to explain its implementation, we will first explain how applications are designed using a space-based distribution approach.

Consider the typical intentions when implementing the functional application logic of distributed software. Developers usually have notions of what we call *business objects*—objects which implement the actual application logic—with

1. data,
2. behavior, and
3. relationships to other business objects.

In the space-based scenario, these properties map to the following concepts:

1. data can be shared via the space,
2. behavior results in manipulation of shared data, which is protected against concurrency conflicts (CORSO uses a transaction model for this purpose) and reacted upon via subscription to change notifications, and
3. relationships are expressed via links between shared data.

In addition, there are often orthogonal requirements for security and failure conditions. Security is a complex topic: for example, data might need to be encrypted, and users might be assigned certain privileges regarding object access in the space. Concrete security demands are specific to the application domain; in chapter 5 of this thesis, a domain-independent security concern (application-level data encryption) will however be shown in greater detail.

Regarding failure conditions, there are two kinds of inherent fault situations in space-based applications, both of which can be caused by network problems or because a site involved in the system (or the middleware software running on it) goes offline: disconnection of a client from the space and disconnection of a part of the space from the rest. In the first case, the client can't access the space at all, no space objects can be added, removed, or manipulated. In the second case, clients will be able to manipulate only a subset of the space objects; the details of this depend on the application implementation and CORSO's replication protocol, which will be detailed in chapter 2.

All these issues apply when implementing the whiteboard application previously described. For it, we propose a space design with two kinds of distributed objects: one for the individual shapes drawn by the users and one for a drawing object acting as a container for the shapes. Figure 1.5 shows a schematic overview of the application and its use of the space: several clients on heterogeneous devices are connected to the space, accessing the drawing object and the shapes it contains. The visualization also shows that the same space can be used to host multiple drawings at the same time.

To detail the whiteboard's requirements, we will use a schema consisting of the issues just introduced—shared data, links, notifications, behavior, fallback, and security:

Shared Data For each drawing, the space contains a single data object with attributes concerning the whole drawing rather than individual shapes.
For each shape, the space contains a structured data object with elements such as coordinates, size, color, and similar shape attributes.

Links Each drawing object has links to all the shape objects comprising the drawing.

Notifications The application components which display the drawing on each user's computer need to be notified whenever a shape is added, removed, or modified. They therefore subscribe to change notifications of the drawing object as well as of the individual shape objects.

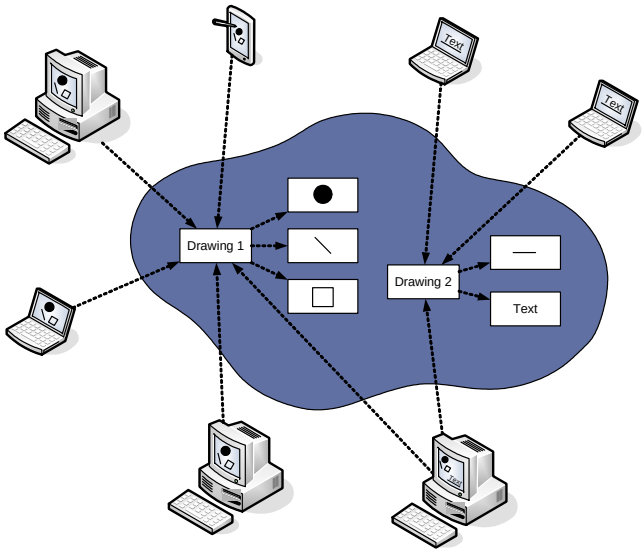

Figure 1.5: Schematic overview over the distributed whiteboard application

Behavior There must be modifier functionality for the individual shapes (change of color, size, coordinates, etc.) as well as for the drawing itself (addition and removal of shapes). For both, concurrent invocation of this functionality by multiple clients could lead to inconsistent, unpredictable, and unwanted results and must therefore be protected by transactions.

Fallback In case a client is disconnected from the space, participation in the drawing process is no longer possible until the client reconnects.
In case part of the space is disconnected from the rest, clients will be able to manipulate only some objects. When the application detects that a manipulation requested by the user is not currently possible, it could react in two ways: inform the user to try again later, or provide an *offline mode*, buffering the changes in the reachable part of the space, until the rest becomes connected again. The former solution is easier to implement, the second solution is probably more user-friendly.

Security Security considerations are ignored for the moment. In chapter 6, we will deal with the requirement of giving different users different privileges.

With this high-level schema, we will now demonstrate how to use CORSO's existing object-oriented language bindings to actually implement these requirements. We will concentrate on one behavioral concern: adding new shapes to the drawing. Since the language bindings demand imperative interaction, this involves the following steps:

1. Create a new space object for the shape,

2. Write the shape data into the new object,
3. Begin a transaction to ensure that concurrent modifications of the drawing are detected and handled correctly,
4. Read the drawing object's data from the space (to be sure to work on the most recent data),
5. Add a reference to the new shape to the drawing,
6. Write the drawing object's data back to the space,
7. Commit the transaction.

In parallel, errors must be detected and handled according to the fallback mechanism.

Listing 1.1 shows concrete code of two methods *AddShape* and *AddShapeReferenceToDrawing*, which implement these steps in $C^{\#}$ using CORSO's .NET language binding *.NET &Co* [Tec04c]. The code demonstrates that even a simple process such as addition of a shape to a drawing can involve many steps if the algorithm is made explicit, rendering an imperative implementation complicated to write and even more complicated to understand.

```
// The Drawing class represents a drawing and contains behavior for
// manipulating the drawing's space representation.
// The class implements the CorsoShareable interface in order to
// allow for drawings to be written to and read from the space.
class Drawing : CorsoShareable {
  private CorsoConnection connection; // represents the space entry point
  private CorsoVarOid drawingObject; // space object for the drawing
  private List<CorsoVarOid> shapeObjects; // links to the contained shape objects
  private int spaceTimeout; // timeout for space operations

  // constructor left out for brevity

  public void AddShape(Shape shape) {
    try {
      // create a new space object for the shape
      CorsoVarOid shapeObject = connection.CreateVarOid();
      // write the shape data into the new object
      shapeObject.WriteShareable(shape, CorsoConnection.INFINITE_TIMEOUT);

      // add a link to the drawing object
      AddShapeReferenceToDrawing(shapeObject);
    }
    catch (CorsoException) {
      // the connection to the space has been lost
      // => inform the user
      HandleDisconnectError();
    }
  }

  private void AddShapeReferenceToDrawing(CorsoVarOid shapeObject) {
    // begin a new transaction to ensure that concurrent modifications of
    // the drawing object are detected and handled correctly
    CorsoTopTransaction tx = connection.CreateTopTransaction();
    try {
      // read the drawing object's data (list of links) from the space
      drawingObject.ReadShareable(this, tx);

      // add a reference to the new shape to the drawing
      this.shapeObjects.Add(shapeObject);

      // write the drawing object's data back to the space
      drawingObject.WriteShareable(this, tx);
```

```
      // commit the transaction
      tx.Commit(spaceTimeout);
    }
    catch (CorsoTransactionException) {
      // concurrent modification detected, try again
      AddShapeReferenceToDrawing(shapeObject);
    }
    catch (CorsoTimeoutException) {
      // drawingObject cannot be reached, the space has been partitioned
      // => inform the user
      HandlePartitionError();
    }
  }

  // Error handlers omitted for brevity
}
```

Listing 1.1: Addition of a shape using CORSO's language binding

For comparison, listing 1.2 shows the same class and *AddShape* method as before, but this time, a more intentional approach is employed by using $C^{\#}$'s mechanism of declarative attributes. The listing shows how this approach could improve the source code both in a quantitative and a qualitative way: it gets shorter and—subjectively—much easier to understand.

```
// The Drawing class represents a drawing and contains behavior for
// manipulating the drawing's space representation.
class Drawing {
  [SpaceLink]
  private List<Shape> shapes;

  // constructor left out for brevity

  [Transactional(ConflictPolicy = TransactionPolicies.Repeat)]
  [PartitionErrorPolicy("HandlePartitionError")]
  [DisconnectErrorPolicy("HandleDisconnectError")]
  public void AddShape(Shape shape) {
    shapes.Add(shape);
  }
}
```

Listing 1.2: Addition of a shape using declarative annotations

Of course, this is only an example of what a declarative approach *could* look like—an actual full declarative middleware interface will be developed in a later chapter. We will also use objective criteria for comparing the imperative and declarative implementations later on, but at a first subjective glance, the comparison shows the motivation of our work: the existing imperative language binding clearly is an inadequate tool, as the implementation based on it contains a high amount of accidental complexity. A declarative version could remedy this by providing a better interface to the network abstraction, subjectively reducing complexity to that minimal amount which is essential due to the requirements.

1.4 Thesis Structure

The rest of this thesis is structured as follows: chapter 2 will give an introduction to the technical background of our work and summarize the state-of-the-art, detailing why current approaches to developing distributed applications are ineffective and insufficient. In the course of doing so, we will cite work by other

authors because of it either being a prerequisite of our work or because it solves similar problems as we do. Chapter 3 will then introduce a thorough analysis of a technology called *runtime-generated subclass proxies*, which we use as an underlying infrastructure for our solution to the problems of efficient[1] space-based application development. The chapter also contains performance considerations regarding this technology, which directly maps to the performance considerations applying to our final solution built upon the infrastructure. With this as a prerequisite, chapter 4 introduces XL-AOF, our aspect-oriented framework to support efficient, declarative space-based application development, which can equally well be used for general-purpose, light-weight aspect-oriented programming. Using this framework, chapter 5 then provides an *aspect-oriented distributed concern library*, the proposed declarative CORSO language binding. The concern library is a catalog of pre-implemented aspect-oriented, declaratively applicable low-level concerns of space-based computing, as well as a number of high-level concerns readily integrable into business applications. Chapter 6 picks up the distributed whiteboard application again, providing a tutorial-like introduction to using the concern library for creating space-based applications. Chapter 7 collects and puts into context the different tables, statistics, and other relevant evaluation data given in the previous chapters, analyzing the actual usefulness of our approach. Finally, chapter 8 concludes the thesis.

[1] Throughout this thesis, we concentrate on *efficient development*, not runtime or memory efficiency. Runtime and memory efficiency of space-based computing has been and will be dealt with by other authors in other works.

Chapter 2

Technical Background and Related Work

In this chapter, we will introduce the background technology and basic technical infrastructure our work is based on. We give an overview of the concepts of space-based computing, including its origins and current implementations, describe the concepts of declarative programming language support for orthogonal concerns—including the concepts for declarative extensibility of modern object-oriented programming languages—, and motivate and define the notions and mechanisms of aspect-oriented programming. In the course of this, we will present and briefly discuss related work dealing with these topics, show some related work more closely related with this thesis, and finally present a matrix comparing the state-of-the-art products and solutions currently available, showing the deficits our approach is bound to solve.

2.1 Space-Based Computing

2.1.1 Linda Tuple Spaces

In 1985, David Gelernter of Yale University presented a new programming language for distributed computing in his paper *Generative Communication in Linda* [Gel85]. Linda is an implementation of an approach developed by Gelernter, in which distributed applications consist of a number of concurrent processes which communicate via an abstract environment called *tuple space*. The tuple space is a kind of distributed shared memory containing *tuples*: collections of passive data values or executable code. Processes communicate by generating tuples (*out* operation), withdrawing tuples (*in* or, more clearly named, *take* operation), and reading tuples without withdrawing them from the space (*read* operation). Gelernter calls this model of communication *generative*, because once generated, each tuple persistently exists until it is withdrawn, accessible to all processes within the space and independent of the process which created it.

Figure 2.1 shows an illustration of a simple tuple space-based distributed application, in which *producer* processes generate tuples and *consumer* processes withdraw them from the space. Together, the processes form a distributed system, with the tuple space being the communication medium.

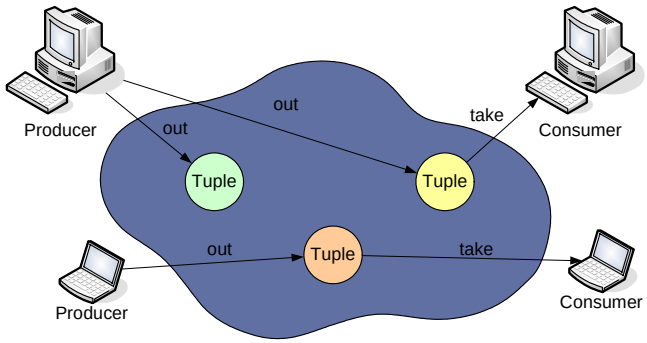

Figure 2.1: Illustration of a simple space-based producer/consumer application

Tuple Types, Names, and Coordination Operations

In the tuple space model, tuples are collections of typed data whose concrete structure is dynamically defined when the tuple is generated. For example, if a tuple should have three members, of which the first is of type *integer*, the second is of type *boolean*, and the third of type *character string*, the respective *out* operation would simply include values of these types (e.g. *5*, *true*, and *"foo"*). When this tuple is to be retrieved via an *in* or *read* operation, the programmer needs to pass along variables of compatible types for each of the tuple's members. That way, the desired data structure is specified both when defining the tuple and when retrieving it from the space. This dynamic typing approach without predefined data schema allows for very flexible programming, but in turn lacks the possibilities of static type checking [Gel85].

Tuples are accessed and identified via textual names. These needn't be unique—several tuples, even with different data structure, of the same name may exist. Each *in* and *read* operation will nondeterministically return a tuple which matches both the structure specification and textual name passed to the operation.

In addition to textual naming, Linda defines a matching mechanism for accessing tuples, which Gelernter calls *structured naming* [Gel85]. For structured naming, *in* and *read* operations specify the values of one or several tuple members in addition to a textual name. The retrieval operations will then only return tuples which match the specified values, the given name, and the structure specification for the remaining members. Table 2.1 demonstrates a few exemplary *in* operations taken from the original Linda article [Gel85].

Inverse structured naming [Gel85] is also possible: *out* operations can define tuples which match any given value passed to an *in* or *read* operation. Examples for this are listed in table 2.2.

Operation	Request semantics
`in(P, i:integer, j:boolean)`	Tuple with name "P", whose members are an integer and a boolean
`in(P, 2, j:boolean)`	Tuple with name "P", whose members are 2 and a boolean
`in(P, i:integer, FALSE)`	Tuple with name "P", whose members are an integer and FALSE
`in(P, 2, FALSE)`	Tuple with name "P", whose members are 2 and FALSE

Table 2.1: Linda *in* operation examples

Operation	Definition semantics
`out(P, 2, FALSE)`	Tuple with name "P", whose members are 2 and FALSE
`out(P, i:integer, FALSE)`	Tuple with name "P", whose last member is FALSE and which matches all requests specifying an integer value as second argument

Table 2.2: Linda *out* operation examples

Coordination and synchronization of the different processes accessing the space is performed via the three space operations, which are executed in an atomic and blocking fashion: if an *in* or *read* operation cannot find a tuple matching the request's arguments, the operation blocks the executing process until another process defines a matching tuple using an *out* operation. This can be used to build powerful synchronization patterns; in fact, a tuple which only has a name is semantically equivalent to a binary semaphore [Gel85]: *in* is equivalent to a semaphore's *P()* operation (processes can only pass if another process signals a *V()* operation), *out* is equivalent to *V()* (allowing waiting processes to pass through *P()*). If a tuple generated by *out* matches blocked *in* requests of multiple processes, one of the waiting processes is nondeterministically chosen, guaranteeing fairness without starvation.

As Gelernter argues in *Linda in Context* [CG89], the coordination language defined by Linda is orthogonal to the remaining programming language: the *out*, *in*, and *read* operations can be embedded in any base language and enhance the distributional capabilities of the language without influencing the remaining language structure. Since the operations are inherently associated with side effects (they are, basically, input/output operations), they are most naturally embedded in imperative languages, but similar to Prolog's I/O predicates [SS86] or Haskell's monad-based *IO* types [BHM02], inclusion in declarative languages is also possible.

Distinguishing Properties

> *«Our goal was to provide communication through space and time. That is, we wanted to allow two processes to exchange information if they were running in different places (different processors or computers) or at different times—one on Tuesday, one on Thursday.»*
>
> N. Carriero & D. Gelernter, "A Computational Model of Everything"

Tuple spaces are distinguished from the more traditional paradigms of concurrent programming *(monitors*, *message passing* and *remote operations* [And81]) because the entity of communication (the tuple) can be written to the space without specifying a receiver, and retrieved from the space without specifying a sender. Gelernter [Gel85] calls this the *space-uncoupling* property of the space.

Tuples can also be written without a process currently waiting for them, and they can be read even if the original generator process is not connected to the space any more, which is called the *time-uncoupling* property of the space.

These two notions of decoupling, which are again described by Beinhart, Murth, and Kühn in 2005 [eKBM05], make the space approach a powerful and distinguished communication paradigm. In their paper "Linda in Context" [CG89], Carriero and Gelernter show a number of concurrent scenarios easily and naturally implemented with Linda (embedded in C), among others a client/server scenario, the *Dining Philosophers* problem [SP88], and DNA sequence comparison. In their 2001 article "A Computational Model of Everything" [CG01], they state that tuple spaces are clearly *not* a model of everything, but they are a natural model of "asynchronous ensembles" of concurrently active processes.

JavaSpaces—A Recent Implementation

While Linda was already devised in the 1980s, the tuple space is still a distinguished and useful paradigm for communication between and coordination of concurrent processes. Therefore, Sun Microsystems included their own version of the tuple spaces, named *JavaSpaces* [Sun03], in their *Java Naming and Directory Infrastructure (Jini)* specification [WT00], according to their web page "an open architecture that enables developers to create network-centric services—whether implemented in hardware or software—that are highly adaptive to change".

Because Jini is a specification centered on Java, JavaSpaces is a version of tuple spaces devised and adapted for the Java platform and integrated into the Java programming language and object model. The JavaSpaces API can be used by referencing a Java class library and contains operations equivalent to those defined by the Linda language. JavaSpaces extends Linda by integrating the space model with Jini's naming and discovery facilities and combining it with Java's serialization mechanisms for easy sharing of objects rather than simple tuples [FHA99].

The whitepaper "Build a Compute Grid with Jini Technology" [Sun04] by Sun Microsystems describes how the space is used for easily implementing a distributed master/worker (producer/consumer) pattern on the Java plaform: masters create computational tasks (units of works) and publish them in the space,

worker processes wait for tasks and execute them (see illustration in figure 2.2). Because of the nondeterminism with several processes waiting for the same kind of objects (and because worker processes start waiting for new tasks as soon as they have finished one), load is equally and fairly distributed within the "computational grid" of participating computers [Sun04].

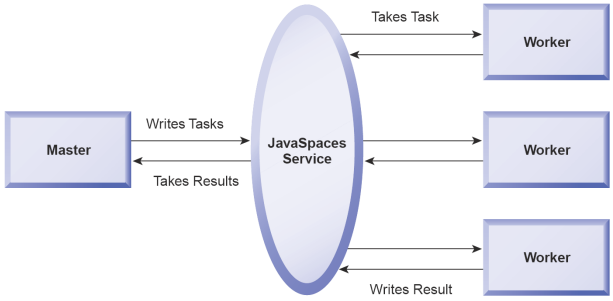

Figure 2.2: Illustration of a master/worker implementation with JavaSpaces (from: *Building a Compute Grid with Jini Technology*, Sun Microsystems, 2004)

Apart from Sun's own JavaSpaces implementation, a well-known implementation is realized by *GigaSpaces*, which extends the JavaSpaces specification, adding distributed implementations of diverse collection interfaces from the Java J2SDK class library [Wel04]. Other Java-based implementations of the tuple space paradigm are *TSpaces* [WMLF98] and *XMLSpaces* [TG01].

2.1.2 CORSO—Coordinated Shared Objects

While the space model as defined by Linda is a convenient abstraction for many distributed scenarios, it has a few shortcomings. For example, tuples have no identity, so references between tuples (e.g. for defining complex data structures) are not possible, neither is updating the content of a tuple without removing it from the space. Similarly, with the Linda model it is not possible for a process to wait for several differently structured tuples at the same time. In addition, although Linda operations are atomic, the approach does not define a mechanism of concurrency protection for operations involving more than one operation.

To overcome these problems, a new space-based approach with newly defined semantics was introduced by eva Kühn in 1994 in her paper "*Fault-Tolerance for Communicating Multidatabase Transactions*" at the Hawaii International Conference on System Sciences. While certainly inspired by Linda, the new approach (first titled *CoK* for "coordination kernel" [eK94] and later renamed to *CORSO* for "coordinated shared objects") provides a full platform-independent, language-agnostic middleware layer for communication and coordination via a shared space. Similar to the Linda model, CORSO processes communicate via shared data objects and are coordinated with blocking operations, but CORSO improves the concept by including peer-to-peer object replication, object iden-

tities, distributed optimistic concurrency control, and near-time change notifications for multiple objects at the same time.

CORSO Space and Object Identifiers

In 2001, Gelernter notes that "a tuple space wasn't centralized on some server; it was distributed over many computers" [CG89], however this was an implementation detail not directly reflected in Linda's tuple space model and space operations. In contrast to that, CORSO has included the concept of peer-to-peer data replication from the very beginning, making it a first-class benefit of the approach. With CORSO, a shared data space is implemented by a number of *coordination kernels* running on different devices connected via a network. Processes participate in the distributed application by connecting to one of these kernels, thus gaining access to the shared data space. Between the different kernels, data is replicated using an intelligent peer-to-peer replication protocol (PR_{deep}), as detailed below, with data only being replicated to a kernel if necessary. In addition, CORSO persists the contents of the space into local databases situated on the devices running the kernel, resulting in easy and reliable automatic data recovery if a kernel or device needs to be (temporarily) shut down [eK94].

Similar to the Linda tuple space a CORSO space contains entities of structured data for information exchange, but, different from tuples, each of these data objects is identified by an *object identifier* (OID), making it uniquely referenceable throughout the whole space. Because of this identity, data objects can not only be created, read, or taken from the space, but they can also be (re-)written, with CORSO objects being distinguished by being writable only once (*constant* objects) or many times (*variable* objects). Read operations on variable objects include a *timestamp*: a numeric version number n indicating that the initiator of the operation requires at least the $(n+1)th$ value written to the object. For example, a process which requires any data from a variable object could specify a read timestamp of 0. However, if the process has already read from an object with timestamp 3 and is waiting for the *next* data written to the object, it would specify a timestamp of 4.

This mechanism of identifiable, rewritable, and versioned objects, which can also link to each other, facilitates the implementation of complex coordination data structures. For example, it allows the creation of linked lists (*streams* [eKN98]) or trees, which—in combination with create, write, and blocking and non-blocking read operations—can then be used to implement more complex coordination patterns, examples of which include the classic *Producer/Consumer* or *Request/Answer* patterns [eK01], the *Connector/Acceptor* design pattern for supporting space access of devices with unreliable network connections [Sch96], and the useful *Replicator* design pattern for replication of heterogeneous data residing on different computers [WeKT00]. As an entry point for these coordination data structures, objects can be assigned names and looked up using these, as in the Linda model. Unfortunately, CORSO does not support an equivalent to Linda's template-based object lookup.

Lifecycle Management

For lifecycle and memory management of coordination data structures, CORSO provides two mechanisms. On the one hand, automatic garbage collection is implemented via reference counting; on the other hand, CORSO provides operations for manually deleting single objects and whole object graphs for optimization. For automatic lifecycle management, the references to CORSO objects held directly and indirectly by processes are counted, and the count is automatically decremented when a process disconnects. When the reference count of an object reaches zero, the object and all connected objects not referenced by other processes are deleted from the space. Named objects always need to be removed manually, because the automatic garbage collector cannot infer when they aren't needed any longer.

Transaction Model

The original Linda operations *read*, *in*, and *out* were defined to be atomic; parallel processes could safely execute them concurrently while keeping the space in a consistent state. However, the processing of more complex data operations, which might consist of multiple operations on different data items, requires a powerful synchronization mechanism.

With the original Linda model, the only way of achieving such synchronization is to implement a lock protocol using a semaphore tuple, as described in section 2.1.1. All processes would have to adhere to the lock protocol and refrain from accessing any tuples guarded by the semaphore after a process has performed a *P()* operation until it conducts a *V()* operation. With complex data structures involving object identities, this is suboptimal for two reasons: first, it does not allow multiple concurrent readers, hindering concurrency where no conflicts would arise [KSUH93]; and second, it requires all processes to know the semantics of all semaphores and behave accordingly. For example, if a process wants to access the tuples A and B and defines semaphore S for this purpose, all processes accessing either A or B must know about and respect semaphore S. If one process doesn't know S or just doesn't respect the protocol, concurrency conflicts will arise.

CORSO therefore implements a mechanism for optimistic concurrency control: the *Flex* transaction model [eKPE92]. With CORSO transactions, all write operations on space objects are performed in an isolated environment of the coordination kernel the process is connected to. Only when the transaction is committed, the kernel obtains a global lock on all objects involved (read or written) in the transaction (following the PR_{deep} protocol, as described in the following section). It then checks if the object values read during the transaction are still valid and, if yes, makes the previously written values globally visible. If not, the transaction fails and the process will have to compensate.

Compensation in case of transaction errors can be performed in different ways, the most simple being to repeat the whole complex operation. CORSO also allows selective removal of single operations from the transaction if they caused a failure, so that the transaction can be tried to be recommitted. Finally, the *Flex* transaction model supports *nested* transactions, with compensation performed

for single sub-transactions of the surrounding top-transaction; however since this is not significant for our work, we will skip a more detailed description of the concept. For details on nested transactions in the *Flex* transaction model, the paper "A Multidatabase Transaction Model for InterBase" [ELLR90] can be consulted.

Replication Protocol PR_{deep} and Persistence

For developing CORSO applications, it is vital to understand the Passive Replication/Deep (PR_{deep}) protocol the middleware uses to replicate data objects between the different devices forming the space. Kühn presents this protocol in the original CORSO paper ("Fault-Tolerance for Communicating Multidatabase Transactions" [eK94]), and we quote an excerpt of the description she gives:

> [E]very site that may access a communication object owns its own copy (replica), with one copy, the main copy, distinguished as the *primary copy*. All other copies are called *secondary copies*. If a communication object is created, a primary copy is created. If the communication object is transmitted to another site, a secondary copy is created and transmitted to the [coordination kernel] at the remote site. Thus, for each communication object i a distributed tree (DT_i) builds up, where each node knows the nodes to which is has transmitted a copy (sons), and each son knows the node (father) from which it has received a new copy. The root of a DT_i is the primary copy of communication object i.

To illustrate this concept, consider the distributed tree shown in figure 2.3 for an object O, whose primary copy is situated within the kernel at host $H1$ (the primary copy forms the root of the tree). When two other hosts $H2$ and $H3$ request a reference to object O from host $H1$ (either by looking up the object's name or by obtaining the reference from a coordination data structure), secondary copies are generated and inserted in the tree below $H1$. Similarly, secondary copies for $H4$ and $H5$ are generated and inserted below $H3$ when these hosts obtain a reference to O via $H3$.

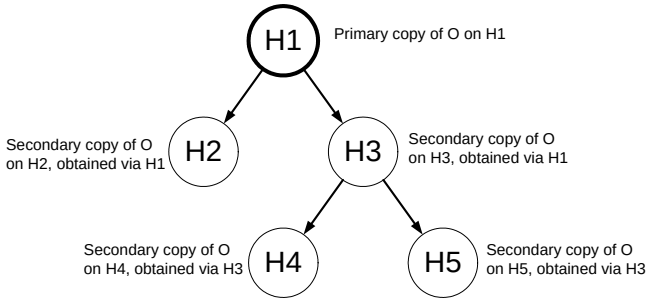

Figure 2.3: Distribution tree of the PR_{deep} protocol example

Only a host which owns the primary copy of an object is allowed to commit a transactional operation on that object. Therefore, if a host tries to commit

a transaction, it first needs to obtain the primary copies of all objects read or written from within the transactional context. For this, a host sends *migration request* messages to its parent nodes in the respective distributed trees, which pass them on to their parent nodes, and so on. When the request reaches the host owning the primary copy, the copy is transferred, unless the owner itself is currently committing a transaction, in which case the request is denied [eK94]. When the primary copy is transferred, this results in the distribution tree being reorganized so that the new owner of the copy is the new root of the tree. For illustration, figure 2.4 shows the transformed tree of figure 2.3 after $H5$ received the primary copy via $H3$ from $H1$. Note that the topology of the tree is not changed by the primary copy migration, only the root node is changed.

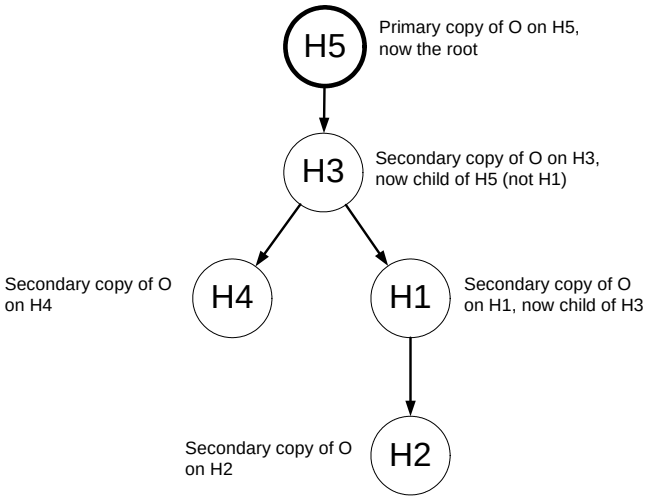

Figure 2.4: Distribution tree after a primary copy migration

CORSO ensures that critical messages for reorganizing the tree are retransmitted as necessary and that all participants always have a consistent view of the tree. If a network connection breaks during reorganization, the distribution tree can always be recovered when the connection is reestablished [eK94].

Besides indicating the current position of the primary copy, the distribution tree also influences how object changes are distributed within the network, depending on the *distribution strategy* of an object. Objects can be set to be either *eager* or *lazy*. In the first case, changes conducted on the primary copy of an object are immediately propagated to all the peers in the distribution tree (always from parent to children), resulting in the different replicas always being held up-to-date.

In the second case, changes are not immediately propagated. Instead, a host will retrieve the lazy objects' data only when a read operation is conducted on the object. For a constant object, the host initiates a search for data via the object's parents, searching up the tree until a host is reached which already holds the object's value. Variable objects, on the other hand, are again distinguished in *read main* and *read next* objects. *Read main* objects will always retrieve the most current data from the holder of the primary copy (searching up the tree from child to parent), whereas *read next* objects use the *timestamp* contained

within the request: a search for data is started up the tree (from child to parent) until a copy is found whose data, is recent enough, according to the request's timestamp. The timestamp and distribution mechanisms do not influence the transaction semantics: if a read request is part of a transaction and the read data is found not to be up-to-date at the time of primary copy migration, the transaction commit fails, no matter what the read timestamp was and whether the object was lazy or eager.

Coordination kernels running the PR_{deep} protocol also provide object persistence: depending on the persistency configuration of a space object (0—no persistence, 1—persistence, 2—redundant persistence), a coordination kernel will save any changes made to the primary copy of an object in a local database. Therefore, when the local system fails and the coordination kernel needs to be restarted, the part of the space known by the kernel can be recovered from the locally saved data [eKN98].

Obviously, the PR_{deep} protocol has beneficial properties because communication between the hosts is organized in a peer-to-peer, or rather child-parent fashion, which results in logarithmic search complexity. It is also a robust protocol: when a host holding a secondary copy is disconnected, the remaining space participants can continue their work. However, it must be noted that the protocol has a weakness when the host holding the primary copy is disconnected: in this case, access to the object is stalled until that host reconnects: there is no way of recovering a primary copy if the host holding it stays disconnected.

Near-Time Change Notifications

While original Linda-style read operations only allow a process to wait for a single space object to be written, with complex collaboration data structures there often comes the need of waiting for different (maybe differently structured) objects at the same time. For this, CORSO provides a concept of change notifications, a notification being a special kind of space object managed by the coordination kernel. OIDs of arbitrary space objects can be inserted into the notification object, and a process can *wait* for the notification to be fired. When the value of one of the objects referenced by the notification is changed, the kernel will automatically send wake-up messages to all processes waiting for the notification.

Change notification messages include the new value of the changed objects and are always sent to all waiting processes without regard to the distribution strategy of an object, so the processes automatically retrieve the most recent data as fast as possible—in *near-time*.

Language Bindings

The coordination mechanisms provided by the CORSO middleware must be interfaced by an application programmer, but CORSO does not define a new programming language. Instead, it provides a set of *language bindings*—application programming interfaces, usually in library form, for common current programming languages. For the most recent CORSO kernel versions, bindings exist for

integration with the Java platform [Tec04b], .NET [Tec04c], and C++ [Tec04a]; earlier bindings were also available for plain C [For95] and PROLOG [eK96]. The bindings provide encapsulations for connecting to a CORSO kernel via a network, for executing coordination operations, for accessing and creating data objects, and for reacting to notifications in a way suitable for the target language.

Because it is important for this thesis, we will shortly describe the details of the language binding *.NET &Co* (the "Co" standing for "Coordination") in this section for reference. We also include the interfaces of these classes in appendix A.

CorsoConnection Class Instances of this class represent a network connection to a CORSO coordination kernel. They provide functionality for establishing and closing the connection, and they act as factories for most other objects in the language binding.

CorsoShareable Interface Classes which need to deposit complex data in a CORSO data object need to implement this interface, which defines *Read* and *Write* methods for deserialization and serialization of space data. The data is serialized into and read from *CorsoData* objects provided by the language binding.

CorsoData Class Provides methods for serializing and deserializing complex data (*structured objects*) to and from the space.

CorsoStrategy Class Encapsulates replication strategy (*eager* or *lazy*), read flags (*read main* and *read next*), and persistence class (0, 1, or 2) of a CORSO data object.

CorsoOid Class This class and its subclasses represent CORSO data objects in the space. They provide methods for reading and writing an object (simple data is read and written directly, complex data is serialized via the *CorsoShareable* interface), naming the object and retrieving its name, accessing its distribution tree, and deleting it from the space. There are subclasses for constant and variable objects, *CorsoConstOid* and *CorsoVarOid*, the latter of which adds timestamp handling. *CorsoOid* objects are created by an instance of *CorsoConnection* either as the result of looking up a named object or as references to completely new data objects.

CorsoTopTransaction Class This class, together with its abstract base class *CorsoTransaction*, allows the execution of complex transactional operations. *CorsoTopTransaction* objects, after having been created by a *CorsoConnection* object, are used to encapsulate primitive space operations, which is done by passing them to the respective OID operations. When the complex operation is fully defined, it can either be committed or aborted. In the former case,

a *CorsoTransactionException* is raised if the transaction cannot be committed due to a concurrency conflict. *CorsoTopTransaction* also provides the option of *trying* to commit a transaction and, if this fails, cancel (and maybe reexecute) those primitive operations which caused the conflict, as indictated by a *CorsoTransFailInfo* object.

CorsoNotification Class Instances of this class encapsulate CORSO's notification facility. They provide *Add* and *Remove* methods for adding OID references (wrapped in *CorsoNotificationItem* objects) to the notification, and a blocking *Start* method, which blocks until a near-time change notification for one of the monitored data objects arrives. In many applicarion scenarios, this method will be called from a dedicated background thread, thus allowing *background notification* without blocking the whole application.

CorsoException Class Because network-based communication is inherently linked to failure, there is a high number of error conditions which may occur while accessing the CORSO space. The methods of the .NET &Co language binding indicate these by throwing *CorsoExceptions* or instances of one of its subclasses. Unfortunately, exceptions meant for user errors (i.e. mistakes made by the API user) are also part of the *CorsoException* hierarchy, which unnecessarily complicates error handling. There exist exception subclasses for different situations in which an error occurs, (such as *CorsoReadException*, *CorsoWriteException*, and *CorsoTransactionException*), but most failure details must be accessed via numeric error codes from the exception object, which, must be noted, is often inconvenient and cumbersome.

CorsoTimeoutException Class All CORSO operations potentially involving remote kernels (i.e. other kernels than those the process is directly connected to) provide a *timeout* parameter. This specifies the number of seconds the connected kernel should spend trying to reach the remote kernels for such an operation. If one of the remote kernels cannot be reached in the given time, a *CorsoTimeoutException* is thrown. This indicates that either the remote kernel cannot be reached or the given value is too low for the current network situation.

2.1.3 XVSM—Extensible Virtual Shared Memory

While the CORSO middleware is a powerful implementation of the space-based computing paradigm, its concepts stem from 1994, at the time of this writing 12 years in the past. Real-world experience has shown need for additional features (such as an additional replication protocol to PR_{deep}, which is optimal for some, but not all cases, as well as Linda-style data matching), and a more *extensible* approach (e.g. with pluggable persistence and and transaction managers) has become desirable. Therefore, in "Improving Data Quality of Mobile Internet Applications with an Extensible Virtual Shared Memory Approach" [eKBM05] Kühn, Beinhart, and Murth first mention a new space-based approach called *XVSM (Extensible Virtual Shared Memory)* as the successor of CORSO, but without speaking of the differences and new features.

In the meantime, the planned feature set of XVSM has been made available as a technical report [eKRJ05], although the full specification has not yet been published. It can already be said that XVSM will provide a sophisticated dual concept, offering a number of often-used coordination data structures preimplemented as so-called *containers*, which in turn can form more complex data structures by being linked together. Within containers, data is stored in classic tuple form and can be accessed using different lookup mechanisms (e.g. indexed access, first-in/first-out access, Linda-style matching, etc.). Containers will support pluggable extension by providing an *interception* mechanism: components can be registered for different interception points, reacting to, modifying, and controlling access to individual containers or tuples.

2.2 Language Support for Orthogonal Concerns

When developing a software application, there often arise requirements (*concerns*) which are said to be *orthogonal* to the main application functionality, including, for example, data distribution, caching, security, fault tolerance, and similar. In this context, orthogonality means that the concerns have no side-effects or dependencies on each other or on the main, functional concerns. Ideally, implementations of such concerns could be exchanged independently and without the rest of the application having to change [Ray03].

In practice, even if high-level concerns such as the ones mentioned above are orthogonal in concept, they almost never are in implementation: concerns and functionality always influence each other in code. Still, implementation and application of orthogonal concerns should be performed as independently as possible, keeping the points of interaction between the concerns at an absolute minimum in order to achieve a decoupled and modularized design. Unfortunately, imperative programming, including the object-oriented programming paradigm, which is by far the most common paradigm for software development at the time of this writing, is not well-suited for implementing and applying orthogonal concerns, because it requires concern functionality to be explicitly and imperatively invoked at (usually) many program points. Because these invocations connect and couple otherwise unrelated classes and methods, they can be seen to *cut through* the application code and design; Kiczales et al denote them as *cross-cutting concerns* in the paper "Aspect-Oriented Programming" [KL+97].

The problems caused by cross-cutting concerns are twofold: on the one hand, modular source code entities (e.g. methods and classes in object-oriented programming languages) meant for encapsulating a functional concern are forced to contain code for the cross-cutting concerns as well, which breaks the "one class, one responsibility" rule considered an important design guideline for object-oriented software [Sut00]. Kiczales et al call this property *tangling* of concerns [KL+97]. On the other hand, code belonging to single cross-cutting concerns is separated and duplicated among different source code entities (methods and classes); this is called *scattering* of concerns [KL+97].

For illustration of scattering and tangling, consider figure 2.5, which is taken from the "AO Tools: State of the (AspectJ™) Art and Open Problems" pre-

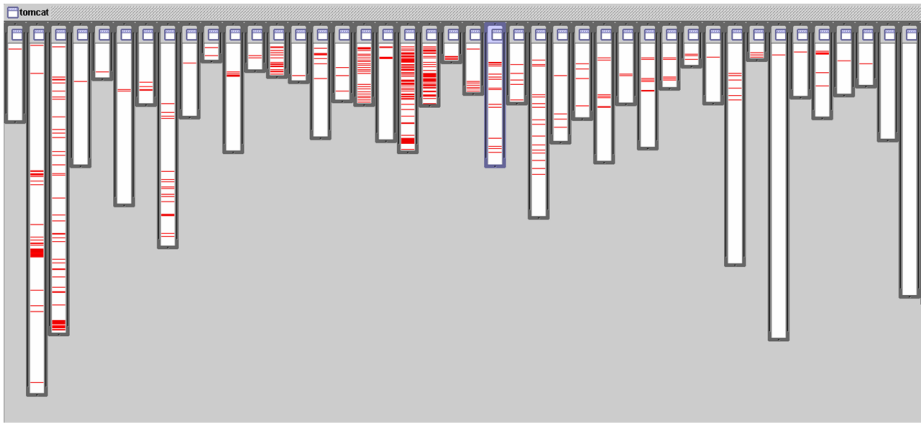

Figure 2.5: Logging code in the Apache Tomcat application server (from: *AO Tools: State of the (AspectJ™) Art and Open Problems*, Mik Kersten, 2001)

sentation held by Mik Kersten at the OOPSLA 2002 AOSD Tools Workshop. It shows an often-cited cross-cutting concern buried within the J2EE open-source application server *Apache Tomcat*: logging [Ker02]. In the picture, logging code is highlighted red, demonstrating effectively how it is tangled with functional code (white) and scattered over many classes and methods.

2.2.1 Built-In Support

There are different strategies to cope with cross-cutting concerns in imperative programming languages. One common approach in research environments is to extend an existing general-purpose programming language and include declarative support for a number of predefined concerns. As two representative examples for this approach, we will very briefly discuss the classic *Argus* system [Lis88] and the more recent *D* framework, both of which constitute research extensions of programming languages with support for concerns for communication and coordination.

The Argus system, devised in the early 1980s by Barbara Liskov and Robert Scheifler, is an extension of the CLU programming language [LSAS77], which in turn is an Algol-based programming language featuring encapsulation mechanisms for abstract data types. As Liskov and Scheifler explain in "Guardians and Actions: Linguistic Support for Robust, Distributed Programs" [LS83], Argus extends CLU with linguistic features such as *guardian* modules, which implement concerns of fault tolerance, and *atomic actions*, which implement concurrency concerns.

Similarly, the D framework presented in 1997 by Cristina Videira Lopes and Gregor Kiczales in "D: A Language Framework for Distributed Programming" [LK97] is an extension of the object-oriented Java programming language. It allows the main functionality of programs to be written in *Jcore*, a Java dialect, and adds a separate new coordination language, *Cool*, for declaratively implementing concerns of distribution and synchronization.

Both systems provide very convenient ways of dealing with the specific orthogonal concerns in a declarative way, but neither of both approaches had significant industrial access. This might well be caused by the fact that both approaches offer support only for the limited number of cross-cutting concerns they explicitly address—the problems caused by cross-cutting still persist for other concerns.

Nowadays, industrial languages also take this approach for dedicated orthogonal concerns: both the Java and $C^{\#}$ languages have built-in declarative mechanisms for thread synchronization through mutual exclusion (*synchronized* in Java [Lea96], *lock* in $C^{\#}$ [ECM05a]), Java also provides language support for serialization control through its *transient* keyword [BP01], $C^{\#}$ has a declarative mechanism for explicit resource cleanup (*using*).

2.2.2 Extensible Support through Extensible Metadata

For both research and industrial programming languages, the lack of extensibility makes language extension for specific cross-cutting concerns a poor way of dealing with the problem. Therefore, recent developments have been about making general-purpose programming languages more extensible by so-called *extensible metadata* mechanisms [Lum02].

With such mechanisms, programmer-defined descriptive items (metadata) are declaratively attached to program source code elements, giving them additional or different semantics to that predefined by the programming language. Metadata is usually not evaluated by the standard compiler for the language, but by additional tools which know and implement its semantics. The first industrially successful programming platform to include this feature was the Microsoft .NET *Common Language Infrastructure* [ECM05b], and all its main programming languages ($C^{\#}$ [ECM05a], Visual Basic .NET [BM02], Managed C++ [Sut02], and Visual J$^{\#}$) included language support for metadata. In 2005, version 1.5 of the Java programming language also introduced metadata, calling it program *annotations* [FF04].

While metadata solves the problem of extensibility, current implementations have an important deficit: they do not provide a convenient and powerful mechanism for specifying the semantics of programmer-defined metadata—metadata, in its current form, is information without behavior, unless explicitly and manually inspected and reacted upon by the application programmer or a dedicated tool.

2.2.3 Custom Attributes—Metadata in .NET

Because the code samples provided in this thesis are mostly given in $C^{\#}$, and the environment introduced in later chapters targets the .NET platform, we will now give a short introduction to $C^{\#}$'s implementation of metadata, called *custom attributes*.

As defined by the "$C^{\#}$ Language Specification", standardized by the European Computer Manufacturer's Association (ECMA-334), attributes are created through the declaration of *attribute classes*, which can have *positional and*

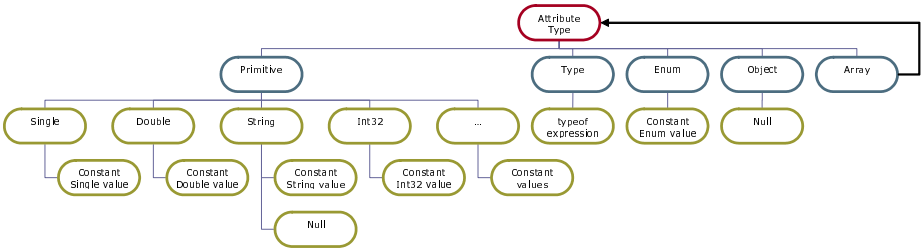

Figure 2.6: Possible attribute argument types and values

named parameters, are associated with program entities using *attribute specifications*, and can be retrieved at runtime as *attribute instances* [ECM05a].

Attribute Class An attribute class is a class which is directly or indirectly derived from the abstract *System.Attribute* base class provided by the .NET Base Class Library. By declaring an attribute class, a new kind of attribute is created, which can be applied to program entities. By default, the attribute can be applied to any kind of program entity, but this can be restricted by applying a *meta-attribute* called *System.AttributeUsage* to the attribute class. The meta-attribute can also be used to control details such as whether the newly defined attribute can be applied to the same program entity more than once.

Positional and Named Parameters The attribute defined by an attribute class declaration can take parameters, which are identified either by their position in the attribute specification or by a name. Positional parameters are defined in the attribute class by definition of a constructor: the parameter list of the constructor is also the list of positional parameters of the attribute. Named parameters are defined in the attribute class in form of public fields or settable properties.

Both positional and named parameters can only be of a small subset of types: they must either be of a primitive type (*string*, *int*, *double*, etc.), of type *Type*, of an enumeration type, of type *object*, or of an array of the aforementioned types. At specification time, the parameter values must be constant or *typeof* expressions, which implies that only primitive, type, enum, array, and null values are allowed. Figure 2.6 illustrates the possible parameter types and values. This restricted type system results in easy serialization of the specified parameter values into the resulting .NET assembly file [ECM05b], but it is also a serious restriction for attribute usage.

Attribute Specifications An attribute is applied to a program entity by putting the attribute class name, optionally followed by a parameter list, between square brackets in front of the program entity. If the attribute class name ends with "Attribute", this last part of the name can be left out in the specification. For the parameter list, one constructor of the attribute class must be picked, and all positional parameters defined by the constructor must be given

a value. Named parameters can optionally be given a value by appending *name = value* expressions to the positional parameter list.

As an example, listing 2.1 defines a new custom attribute for describing program entities by defining an attribute class *DescriptionAttribute*. It restricts the attribute's usage on class and method declarations, with only one application per entity allowed, by using the meta-attribute *System.AttributeUsage*. The attribute defines one positional parameter, *description*, and one optional named parameter *SeeAlso*. The attribute is then used to describe a sample class *C* and its members.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
  AllowMultiple = false)]
class DescriptionAttribute : Attribute {
  private string description;
  private string[] seeAlso;

  public DescriptionAttribute(string description) {
    this.description = description;
    this.seeAlso = new string[0];
  }

  public string Description { get { return description; } }

  public string[] SeeAlso {
    get { return seeAlso; }
    set { seeAlso = value; }
  }
}

[Description("This is a class for demonstrating attribute usage.")]
class C {
  [Description("Does nothing, actually.", SeeAlso = new string[] {"Bar"})]
  public void Foo() { }

  [Description("Does nothing either.", SeeAlso = new string[] { "Foo" })]
  public void Bar() { }
}
```

Listing 2.1: Attribute definition and usage sample

In some cases, it is not clear from the context which program entity an attribute specification is targeted at. In such situations, the target can be specified explicitly by preceding the attribute name with string such as *assembly:*, *module:*, *method:*, *return:*, and similar.

Attribute Instances The attributes applied to a program entity can be inspected at runtime using .NET Reflection [Ric06]. For example, the attributes defined on a class or method can be extracted using the predefined reflective *Type.GetCustomAttributes* or *MethodInfo.GetCustomAttributes* operations. When the attributes for a specific program entity are requested for the first time, the .NET runtime will create one instance of the respective attribute classes for each attribute specification associated with the entity, and return the instances as objects to the requester, who may then use them like any other object created with *new*.

2.2.4 Source Code Annotations—Metadata in Java 1.5

Although we will not use it in this work, Java's metadata mechanism defined by the document "JSR 175: A Metadata Facility for the Java™ Programming

Language" [B+04] could be used to implement a Java port of our declarative approach. Therefore, we will include a short introduction to the mechanism for comparison with $C^{\#}$'s implementation.

In the Java programming language, annotations are defined through abstract *annotation types*, with parameters being expressed as method declarations of these types. Annotations are applied to program elements via *annotation modifiers* and can be inspected at compile-time, load-time, or runtime, as defined by their *retention policies*. Sun's Java virtual machine distribution also includes an *annotation processor tool*, which can be used to perform code transformations based on annotations.

Annotation Types Annotation types are a special kind of interface, declared via the *@interface* keyword. By declaration of an annotation interface, a new annotation is defined and can be applied to program elements. The program element kinds to which the annotation can be applied are specified via the *meta-annotation @Target*, which is predefined by the Java class library.

Annotation Type Methods Like ordinary interfaces, annotation types can declare methods (without specifying implementation). The methods of annotation types, however, represent the parameters of the declared annotation; therefore their signature is restricted: methods of annotation types must not be generic and are not allowed to take any parameters. The methods' return types specify the types of the annotation's parameters and are restricted to primitive types, *String* or *Class*, an enumeration, an annotation type, or an array of these types. It is also possible to specify default values for an annotation parameter—making it optional at annotation application time—by appending a *default* clause to the respective method declaration.

The method names define the parameter names of the defined annotation. If an annotation should only have one parameter, the respective method should be called *value*.

When an annotation is instantiated via reflective access to a program element, an automatically generated implementation of the interface is returned, whose (parameterless) methods return the parameter values specified within the annotation modifier attached to the program element.

It has to be noted that in comparison to .NET attributes, annotation interfaces are somewhat restricted because as interfaces they cannot have any methods with user-defined behavior. When an instance of a .NET attribute is retrieved, it is a fully featured object, wheres an instance of a Java annotation is merely a container for user-defined values. On the other hand, annotation interfaces allow the programmer to pass annotations as the parameter of another annotation, thus enabling the programmer to apply tree-like annotation structures to program elements—a feature missing with .NET.

Annotation Modifiers Annotations are applied to a program element (such as classes, fields, or methods) by means of an annotation modifier, which consists of the symbol @ followed by the name of the annotation type and the

annotation's parameter list. The parameter list can be omitted for annotations whose interfaces contain no methods (or only methods with default values); for annotations with only one parameter, this parameter is simply written between parentheses. For more complex annotations, the parentheses contain comma-separated *name* = *value* pairs, where *name* must correspond to one of the methods of the annotation interface. The parameter order is not significant, but all methods without default values must be specified, with *null* references not being permitted as values regardless of the parameter type.

As an example, listing 2.2 shows an *@Description* annotation serving the same purpose as *DescriptionAttribute* in listing 2.1. The annotation is then used to describe a class C and its members.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@interface Description {
  String value();
  String[] seeAlso() default {};
}

@Description("This is a class for demonstrating annotation usage.")
class C {
  @Description(value = "Does nothing, actually.", seeAlso = {"Bar"})
  public void Foo() { }

  @Description(value = "Does nothing either.", seeAlso = {"Foo"})
  public void Bar() { }
}
```

Listing 2.2: Annotation definition and usage sample

Retention Policies The Java class library provides the meta-annotation *RetentionPolicy*, which is applied to an annotation type in order to specify whether the annotation should be retained in the source file only (i.e. the annotation can only be retrieved at compile time), in the class file as well (i.e. the annotation can be retrieved at compile-time or at load-time), or even for runtime representation (so that it can be retrieved at any time, including runtime).

Annotation Processor Tool The Java 5 SDK provided by Sun Microsystems includes an annotation processor tool, which provides a number of reflective and code generator classes for defining program transformations based on annotation specifications as a build step when compiling the program. Unfortunately, the tool's object model is somewhat restricted. For example, it is not possible to inspect a method's source code from within the program transformer. Third-party transformation tools such as SPOON [Paw05] are therefore preferable for nontrivial transformations.

2.3 Aspect-Oriented Programming

In 1997, Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar, a research team

of the XEROX Palo Alto Research Center published a paper titled *Aspect-Oriented Programming* [KL+97] at the European Conference on Object-Oriented Programming. According to the authors, the—at that time—most widespread methodologies for software development suffer from a common deficit. Procedural and object-oriented programming inherently support separation of concerns—the process of breaking a program into distinct non-overlapping units of encapsulation—only in one dimension. Using these paradigms, software problems are always decomposed in one direction.

This property, which was later coined as the *tyranny of the dominant decomposition* [TOHSMS99], makes the mechanisms of encapsulation provided by these paradigms, i.e. procedures, modules, and classes, very well-suited for decomposing (hierarchically structured) functional concerns. However, it makes them equally ill-suited for encapsulation of orthogonal concerns, whose decomposition, by definition, does not follow the same dimension. Integration of orthogonal concerns into procedural or object-oriented programs automatically makes them cross-cutting concerns—concerns which cut through the functionally decomposed classes, modules, and procedures.

As a solution to this dilemma, Kiczales et al suggested to apply a new methodology of decomposition in parallel to the existing ones: like classes, which encapsulate and decompose functional concerns, *aspects* should be employed to encapsulate cross-cutting concerns. According to the research team, such encapsulations should be similar to classes, but include an additional mechanism for specifying the points of interaction between the cross-cutting and the functional concerns, the so-called *join points*. A system executing or compiling an aspect-oriented program would automatically link the well-encapsulated implementations of the functional and cross-cutting concerns together at these points as defined by a set of rules (the so-called *weaving process*), resulting in a fully composed and thus fully functional program. Figure 2.7 illustrates this approach schematically.

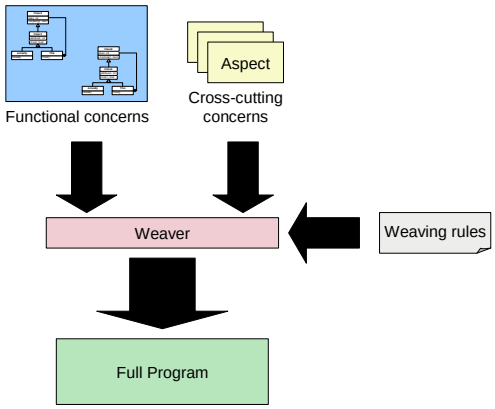

Figure 2.7: AOP weaving process

While weaving is usually done at compile time (including the possibilities of

pre- or postcompilers) or load-time, it is also possible to implement weaving at runtime [CS06]. Runtime weaving, also known as *dynamic weaving*, allows aspects to be added and removed at runtime, which is beneficial especially in mobile and long-running (24/7) applications, which cannot be easily redeployed, or even recompiled, when a change in aspect configuration is needed. Dynamic AOP still has some weaknesses, though; a clean semantic model is missing (what happens if an aspect is currently executing when it should be removed?), and implementation can be hard without support by the underlying platform [PAG03].

2.3.1 AspectJ—De Facto Reference Implementation

Although there are many implementations of the AOP paradigm for different platforms (e.g. JBoss AOP [FR03], Spring AOP [JH+05], or JAC [PSDF01] for Java, AspectS [Hir03] for Smalltalk, AspectR for Ruby), there is one de facto reference implementation, which is an aspect-oriented extension of the Java platform: *AspectJ* [tAT05]. Its popularity probably stems from three reasons: it was the first AOP tool which implemented the criteria defined in the original "Aspect-Oriented Programming" paper, with Gregor Kiczales being among the founding members of the project [LK98], it has by far the most exhaustive feature set of aspect-oriented implementations currently available, and it is based on the industrially successful general purpose programming language Java [PSR05].

AspectJ picked up the terms defined by Kiczales et al [KL+97], giving them more specific semantics in a concrete implementations. The terms and notions used for AOP research are therefore mostly identical with the language constructs defined by AspectJ, and we will also use these terms in this thesis. Their semantics is defined by "The AspectJ 5 Development Kit Developer's Notebook" [tAT05], "The AspectJ Programming Guide" [tAT03], and other sources [PSR05], and we will describe it in the following sections. It has to be noted, however, that the definition of AOP is still somewhat flexible, and even AspectJ can't be said to implement *the* AOP paradigm, because the paradigm is still being researched and evolving.

Aspects, Join Points, and Join Point Shadows

An aspect is a modular unit for encapsulation of a cross-cutting concern. Like classes, aspects can have methods, fields, and constructors, but unlike classes, they can have *crosscutting* members, as explained in the subsequent sections. To form a concrete program, aspects are woven with *target code* (also called *base code*), influencing both its static structure and dynamic behavior. The concrete program points during dynamic program execution where aspects change the behavior of the target code are called *join points*. The locations reflecting these points in the static program source code are called *join point shadows*.

Aspects can be instantiated, and as with ordinary objects, each aspect instance contains one independent set of the instance fields defined by the aspect. In contrast to ordinary object instantiation, however, aspect instantiation is not

performed explicitly by a programmer (e.g. using a *new* expression), but automatically within the context of the target code associated with the aspect. In AspectJ, aspect instantiation can follow different policies: *singleton* (there exists one aspect instance per application), *per type* (aspect instances are created for and associated with specific target classes), *per object instance* (aspect instances are created for and associated with object instances in the target code), and *per control flow* (aspect instances are associated with the control flow of target code, e.g. to a specific method call sequence). Other, more fine-grained and user-controlled policies (e.g. per join point or per group of object instances), are also conceivable.

Advice, Pointcuts, and Introduction

When an AOP tool weaves together aspect code and target code, there are three issues that need to be taken into account:

- At which join points does the aspect influence the behavior of target code?
- In what ways does the aspect influence the behavior of target code?
- In what ways does the aspect influence the static structure of the target code?

These questions are dealt with in AspectJ using the three notions of *pointcuts*, *advice*, and *introduction*, in that order.

Pointcuts select a subset of the available join points in the program. For this, AspectJ defines a dedicated pointcut language, which allows to quantify over the program elements, e.g. via specifying class and method names (including wildcards), parameter values, and control flows (e.g. method call sequences). Pointcut expressions can be combined using boolean *and*, *or*, and *xor* operators. AspectJ's pointcut language is very verbose, although (subjectively) not very readable, and pointcut expressions can easily become very complicated. It is therefore both possible and sensible to use simple *named pointcuts*, i.e. pointcut expressions assigned a name, and combine those in order to form more complex, but still understandable pointcut expressions.

Advice specifies the behavior executed by an aspect at the join points selected by a pointcut expression. AspectJ classifies the behavior into *before*, *after*, and *around* advice, where the first is executed just before a join point and the second just after it. The third is executed *in place of* the join point, and the advice can freely decide whether to proceed with the join point or not. The *after* kind has the extensions *after returning*, which only occurs if a join point returns a value (i.e. throws no exception), and *after throwing*, invoked when the join point throws an exception. While this distinction is definitely useful, it is not compatible with all join point kinds (e.g. an *after returning* advice can only be attached to method call join points, but not, for example, to field access join points). Therefore, some approaches apply the classification to the join points instead of advice (e.g. having *before method call*, *after method call*, and *around method call* join points instead of just one *method call* join point).

In AspectJ, advice code can also get contextual information from the associated join points. For example, an advice woven to a method call join point has access to the parameters passed to the method, to the object making the call, and to the object receiving the call. This is an important feature, because cross-cutting concerns are seldom completely decoupled from the remaining code—most often, they need access to such contextual information.

Finally, introduction, also called *inter-type declaration*, allows aspects to change the static structure of their target code. Aspects can add fields and methods to their target objects, change the base types of classes, and add interfaces and interface implementations to their target types.

Together, these three notions allow effective encapsulation of cross-cutting concerns, and while the recent version 5 of AspectJ has changed some details about the AspectJ language [tAT05], the base features are still consistent with the notions introduced in the original AOP paper.

2.3.2 Definition of AOP

While the paper of Kiczales et al [KL+97] is considered the origin of aspect-oriented programming, and AspectJ is considered a de facto reference implementation, the *essence* of AOP has yet to be defined. Filman and Friedman tried to do so in their much-quoted paper "Aspect-Oriented Programming is Quantification and Obliviousness" [FF00], where they define AOP as follows:

> *«[We propose] that the distinguishing characteristic of Aspect-Oriented Programming (AOP) systems is that they allow programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions. [...] AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.»* Robert E. Filman and Daniel P. Friedman

Quantification, in this context, refers to the ability of saying: "In programs P, whenever condition C arises, perform action A" [FF00]. According to the authors, this makes three dimensions of classification available for AOP tools: *quantification*, i.e. what set of conditions C the tool allows to be specified (this comprises the *join point model*'s feature set as well as the pointcut language), *interface*, i.e. how the actions A interact with the target code and with each other, and *weaving*, i.e. how the system is made to intermix actions A and P. *Obliviousness* refers to the property of the programmer not being aware of what cross-cutting concerns affect a class (and the class not being specifically designed to allow for the concerns to affect it).

While quantification is indisputably one of the most important features of aspect-oriented programming, the demand for obliviousness has evoked some controversy. Most notably, Sullivan, Griswold, et al have performed an extensive case study showing that obliviousness of the base code, while desirable, leads to very awkward aspects full of ad-hoc pointcut expressions, making the cross-cutting concerns hard to encapsulate and the aspects unsuitable for reuse [SG+05]. Instead of oblivious base code, the authors propose that base code

needs to adhere to certain design guidelines in order to make them "aspect-friendlier", enabling clean encapsulations and reusable aspects [GS+06].

2.3.3 AOP vs. Interception

From the (undisputed) definition that AOP needs a mechanism for quantification comes an important implication: AOP is not the same as *interception*, which it is often confused with.

According to Brenda M. Michelson's article "Service-Oriented World Cheat Sheet: A Guide to Key Concepts, Technology, and More" [Mic05], interception is a type of collaboration in which an entity intercepts a request to (or a reply from) a target, forwarding the request (or reply) to the original target if necessary, providing additional behavior in an othogonal fashion. Michelson also states that "interception is used to perform common functions such as security, policy, audit, and translation" and that "in many interception scenarios, the requesting and providing parties are unaware of the intermediary service."

Thus, at first glance, interception and AOP have much in common: both are used to implement orthogonal services, both are sometimes applied to oblivious targets. And indeed, different implementations of interception (such as service interception or method interception) can be used as an infrastructure for AOP in different surroundings. However, the important difference between AOP and interception is the notion of *quantification*, i.e. a mechanism or language for quantifying over program conditions in order to invoke cross-cutting code. This is an important part of AOP, but it is not a property of interception.

2.3.4 AOP vs. Extensible Metadata

Extensible metadata implementations like Java annotations or .NET custom attributes allow to orthogonally extend a programming language by new declarative mechanisms, so are they the same as aspect-oriented programming? In fact, metadata and AOP actually have quite different goals—the former is meant to provide a flexible language extension mechanism, while the latter is a paradigm supporting encapsulation of cross-cutting concerns. However, declarative metadata and AOP can complement each other in two important and useful ways.

Firstly, extensible metadata can be used to implement an *aspect language* (i.e. language constructs allowing to define aspects, associate them with join points, etc.) on top of an existing programming language in a standardized, portable way. AspectWerkz [Bon04] was the first AOP tool to do so—its aspect language is set on top of the Java programming language and is implemented completely with Java annotations. (In fact, AspectWerkz had this even before Java included an extensible metadata mechanism—AspectWerkz analyzed an application's source code comments for annotations in earlier versions.) Following its example, AspectJ, while traditionally being based on an aspect-oriented Java language extension (and thus requiring a dedicated compiler), now also includes a purely metadata-based variant of its aspect language in its recent version 5 [tAT05].

Secondly, metadata can be used to declaratively describe program elements within base code. These descriptions can be used by an aspect-oriented tool for join point selection. As an example, a logging aspect might be defined to influence the behavior of all methods tagged with a *Logged* attribute in a certain class. This is discussed by Mezini and Kiczales in "Separation of Concerns with Procedures, Annotations, Advice and Pointcuts" [KM05], and the authors conclude that this is indeed a good way of selecting join points, which can often replace unstable and unreadable enumerative pointcut expressions (which simply list a number of target methods by name). They also state, however, that it is often desirable to have *semantic* annotations, which are not bound to a certain aspect. The example of a *Logged* annotation is clearly bound to the logging aspect. According to Mezini and Kiczales, a better aspect-oriented design would be to find common properties connecting the methods to be logged. For example if all business methods should be logged, a more semantic *Business* attribute could be applied to those, and the logging aspect could then be tied to this *Business* annotation.

2.3.5 AOP Infrastructure Classification

As described earlier, Filman and Friedman noted that aspect-oriented tools could be classified by their quantification abilities (or join point model), interfaces between target code and aspects (as well as aspects among each other), and their weaving model. This classification model is good for full AOP tools, but it is not fully satisfactory when classifying AOP *infrastructure*—platforms, toolkits, or frameworks providing a weaving mechanism which can be used to build an AOP tool.

In recent work [eKFS06], we ourselves have therefore published a similar classification model for AOP *infrastructure* based on the Microsoft .NET platform (although it could be easily mapped to the Java platform). Since this schema will be used for classifying the AOP infrastructure presented in chapter 3, a description of the classification structure will be given in this section.

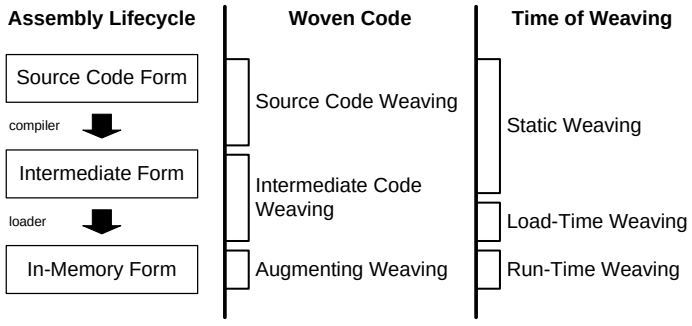

Figure 2.8: Classification of weaving mechanisms

Figure 2.8 shows two orthogonal classifications of weaving mechanisms with regard to an application component's (.NET assembly's) lifecycle. On the one hand, weaving is classified based on the kind of woven code: *source code weaving*

manipulates an assembly's source code to inject aspect or glue code[1]; it can be performed either by a code transformation tool or by a dedicated compiler (plugin). *Intermediate code weaving* manipulates the intermediate code generated by .NET-based compilers (consisting of executable code in the *common intermediate language CIL* or *IL* and type information called *metadata* [ECM05b]) to inject aspect or glue code; this is usually done by a post-compiler or custom class loader. *Augmenting weaving* does not manipulate existing code, but instead augments its in-memory form with new glue code, connecting it to aspect code; this can be performed by frameworks rather than tools. Traditional approaches like AspectJ are source code or intermediate code weaving mechanisms or mixes thereof, but recently, augmenting approaches have drawn much attention [Joh05][eKS05b][CS06].

On the other hand, weaving can be classified based on when it is performed: AspectJ, for example, has traditionally been a static weaver, performing the combination of aspect and target code before the application is loaded into memory [tAT05]. Load-time weavers do that combination just as the application is loaded, and runtime approaches weave while it is actually executing.

With regard to the kind of code being woven, we characterize based on the following properties, displayed in table 2.3 with advantageous properties in boldface:

Invasiveness is a measure for the degree of manipulation the weaving approach performs on user-written code. Source code weaving approaches compiling a dedicated aspect language have low invasiveness, augmenting approaches only extend and also have low invasiveness. Other approaches change the structure of user-written code and are thus highly invasive.

Debuggability denotes how much effort is needed to make the woven program debuggable with standard mechanisms (e.g. Microsoft Visual Studio). This is easy with augmenting approaches, because the original debug information remains valid after weaving. Source code weaving also results in correct debug information. With intermediate code weaving, debuggability involves manipulating a debugger-specific file format. This is not portable and often hard: for example, the undocumented *Program Database* file format used by Visual Studio cannot be easily manipulated.

Join point model denotes the join point kinds an approach can provide. Source code weaving makes no restrictions whatsoever to the join point model. With intermediate code weaving, the only restrictions are those posed by IL and metadata (e.g. there are no "for" loops available in IL; to a certain extent this can be overcome by pattern matching as it is also done by decompilers). Augmenting weaving relies on the manipulation mechanisms provided by the .NET environment, i.e. OOP techniques such as interface implementation and method overriding.

Design prerequisites describes prerequisites needed from the perspective of the application designer. With augmenting weaving, it is often necessary to use a factory which performs the in-memory weaving when objects are instantiated. With source code and intermediate code weaving, there is no such restriction.

[1] *Glue code* is code which "glues" an aspect to its target objects.

Tool prerequisites describes the tools needed for the approach. Source code weaving needs a precompiler or real compiler, intermediate code weaving needs a postcompiler or class loader, augmenting weaving can be done by a framework or library.

Implementation effort is a measure for the effort needed to create a tool based on the approach and keep it up to date with platform changes. Source code weaving requires the most effort by an implementer: it needs at least a source code parser and code emitter. If the tool is to be a full compiler, complexity is even worse. With intermediate weaving, an IL and metadata parser and emitter are needed, although IL is typically simpler to parse than source code. Augmenting weaving only requires a very simple framework.

Compatibility is a measure for the compatibility of the approach with third-party compilers, frameworks, or metaprogramming tools. With compiler-based source code weaving tools, third-party compilers cannot be used. Intermediate code and augmenting weaving tools pose no compatibility problems and can usually be easily combined with any compiler, framework, or tool.

Language support denotes the number of supported programming languages. While intermediate code and augmenting weaving strategies can handle all programming languages targeting .NET, a source code weaving tool can only target a single programming language. Since one of .NET's goals is to be a multi-language environment [ECM05b], this is an important restriction which might exclude a high number of potential users (much more important than on the Java platform, which has only one dominant language).

Performance is a measure for the runtime efficiency of the approach. With source code weaving, performance is optimal, all compiler optimizations and JIT (just-in-time, i.e. runtime compilation) optimizations can be performed. With IL code weaving, compiler optimizations should be disabled in order to retain a powerful join point model (e.g. target methods must not be inlined by the compiler), but JIT optimizations can be performed without any restriction. With augmenting weaving, some optimizations are disabled by the use of certain OOP features (like virtual method calls), but most JIT optimizations are available.

	Source	Intermediate	Augmenting
Invasiveness	**low** to high	high	**low**
Debuggability	**no-effort**	hard	**no-effort**
Join point model	**arbitrary**	**IL and metadata**	OOP
Design prerequ.	**none**	**none**	factory
Tool prerequ.	compiler	postcompiler	framework
Impl. effort	very high	high	**low**
Compatibility	low	**high**	**high**
Language support	one	**all**	**all**
Performance	**optimal**	**good**	medium

Table 2.3: Weaving approaches by code form

With regard to the time of weaving, we characterize the approaches as follows, summarized in table 2.4:

Changeability denotes how much effort is needed to add or remove an aspect to or from the application. With static weaving, recompilation or reinstrumentation of the assembly is needed, the application has to be restarted and redeployed. With load-time weaving, the application domain needs to be reloaded,

often requiring a restart of the application. With runtime weaving, changes can be applied immediately.

Deactivating aspects is equally possible in all three weaving variants and requires some sort of join point manager which is asked before a join point is triggered.

Error detection refers to the point of time when weaving configuration errors are detected. With static weaving, this is before application deployment, whereas it is after deployment with the other two approaches.

Testability is inversely proportional to the effort needed to test an object in scenarios with different (or no) aspects attached to it. This follows directly from changeability: static and load-time weaving require much effort, whereas runtime weaving does not.

	Static	**Load-Time**	**Runtime**
Changeability	recompilation	reload	**immediately**
Deactivating aspects	**immediately**	**immediately**	**immediately**
Error detection	**before depl.**	after depl.	after depl.
Testability	low	low	**high**

Table 2.4: Weaving approaches by time of weaving

2.3.6 Infrastructure Classification in Context

The scheme presented in the previous section is used to classify infrastructure, i.e. weaving mechanisms, not AOP tools. Therefore, its classification model differs from that of Filman and Friedman. We differentiate infrastructural mechanisms based on the form of woven code and the time of weaving, which includes Filman and Fiedman's join point model, but in addition comprises other properties as well. For example, the important feature of *changeability*, i.e. how much support an AOP tool can provide for adding and removing aspects to/from an application, is completely missing in the original classification mechanism. On the other hand, our classification ignores pointcut languages and interface, because these are properties of AOP tools rather than infrastructure.

2.4 Summary of the State-of-the-Art

After having given an introduction to the base technology of this thesis and work related to that, we will now take a closer look at existing approaches tackling the same problems we are addressing in our work. In this thesis, we aim to give an easily adoptable mechanism for more efficient distributed application development by combining the concepts of space-based computing, declarativity in object-oriented programming languages, and aspect-oriented programming, and to the best of our knowledge, we are the first to do so. However, there is work by others that has similar goals as we have or provides similar solutions as we do.

Regarding our goal of facilitating distributed application development by employing the declarative facilities of modern programming languages and aspect-

oriented programming, the open-source J2EE (Java 2 Enterprise Edition) application server *JBoss* [FR03], as well as the recent *Enterprise Java Beans (EJB)* specification version 3.0 [D+06] are related work in that they both employ Java 5 annotations for specifying enterprise concerns on so-called *plain old Java objects (POJOs)*. JBoss also allows aspect-oriented development of application components via its aspect-oriented mechanism *JBoss AOP*. At a closer glance, however, both JBoss and the EJB 3.0 specification actually address different kinds of systems: J2EE is a container-based architecture targeted at large-scale client/server enterprise applications, possibly incorporating publish/subscribe systems. The space-based paradigm on the other hand postulates a lightweight form of distributed computing, with stateful, data-driven communication instead of message passing, with flexible peer-to-peer solutions instead of client/server architectures. Our solution is also different because we provide a lightweight, easily deployable and adoptable framework-based approach to AOP, whereas JBoss AOP is part of a large and heavy-weight application server.

The *Spring* framework [JH+05], on the other hand, and its aspect-oriented component *Spring AOP* as well as its .NET port *Spring .NET* [JP+06] go more into this direction, aiming to provide a lightweight framework for J2EE software development based on *inversion of control* technology. Their goal is to remove much of the accidental complexity introduced by large-scale J2EE application servers and thus to significantly facilitate enterprise application development. Indeed, Spring's approach to AOP is similar to ours, but where we aim for easy adoptibility and ease of use, leveraging built-in declarativity and extensible metadata, Spring aims for adaptability and makes heavy use of XML-based configuration. Also, Spring is still based on J2EE, providing no support whatsoever for space-based technology.

Of the existing implementations of space-based technology—e.g. Linda, JavaSpaces, CORSO, and XVSM—none provides a declarative language interface and none provides support for encapsulating cross-cutting concerns by means of aspect-oriented programming. In the thesis "Distributed Shared Memory in Modern Operating Systems" [sS04], Tomáš Seidmann of the Slovak University of Technology, Bratislava, also describes somehow related work: he implemented a completely new space-based system for distributed computing, including a .NET API which does make minimal use of .NET's custom attributes. Seidmann, however, has his focus more on the implementation of the distributed shared memory than on an intuitive declarative language binding. And again, he does not provide a mechanism for general encapsulation of cross-cutting concerns.

The research-based approaches *Argus* and *D*, described earlier in this chapter, differ from our approach in several ways: while both of them provide a declarative interface to the concerns of distribution—and *D* does so in a lightweight fashion—, they implement it via language extensions rather than extensible metadata; they do not allow to encapsulate additional cross-cutting concerns, and they were not designed for easy adoptability.

Lastly, there are quite a few aspect-oriented environments already existing. We already introduced AspectJ, which—being compiler-based—is not a lightweight approach and, due to its complexity, hard to adopt; but in addition, *NAspect* [Joh05] should be included, because it is a lightweight and adoptable AOP

implementation for .NET not unlike our own. Unfortunately, neither AspectJ nor NAspect include any support whatsoever for distributed (not to mention space-based) computing; NAspect, like Spring AOP, also relies heavily on XML rather than declarativity through extensible metadata.

	Distr	SBC	Decl	MD	AOP	LW	Adopt
JBoss	x	-	x	x	x	-	-
EJB 3.0	x	-	x	x	-	-	-
Spring	x	-	x	-	x	x	-
Linda	x	x	-	-	-	x	x
JavaSpaces	x	x	-	-	-	x	x
CORSO	x	x	-	-	-	x	x
XVSM	x	x	-	-	-	x	x
DSM	x	x	-	-	-	-	-
Argus	x	-	x	-	-	-	-
D	x	-	x	-	-	x	-
AspectJ	-	-	x	x	x	-	-
NAspect	-	-	-	-	x	x	x
Our approach	x	x	x	x	x	x	x

Table 2.5: State-of-the-art feature matrix

Table 2.5 summarizes the properties of the state-of-the-art approaches given in this section and chapter, graphically presenting their feature set and shortcomings. This makes fairly clear that none of the approaches fulfills all the requirements we found for our work; none of the approaches supports distribution (*Distr*) through the mechanism of space-based computing (*SBC*), provides a declarative language interface (*Decl*) through extensible metadata (*MD*), allows for encapsulation of additional cross-cutting concerns via lightweight (*LW*) *AOP*, and was designed with adoptability (*Adopt*) in mind. This is where our approach fits in.

Chapter 3

An Underlying AOP Infrastructure

Before describing the aspect-oriented tool we developed to fulfill the requirements of our work, we will give a detailed analysis of the infrastructure we base our approach on. Using an existing infrastructure for implementing an aspect-oriented tool has the advantage that one can concentrate on the high-level layer of the tool, i.e. how the tool presents itself to the developer, rather than having to deal with the low-level details of implementing an aspect weaver.

In this chapter, we will therefore describe the mechanism of *runtime-generated subclass proxies* as an infrastructure for AOP. Runtime-generated subclass proxies are not a new concept, they have already been used by AOP tools such as JAC [PSDF01] or NAspect [Joh05] as well as non-AOP tools such as NHibernate [Har05]—the novelty of our approach is concentrated on a higher level. We were, however, to the best of our knowledge the first to publish a thorough analysis of the subclass proxy concept with regard to AOP and the first to augment it with the new *dynamic method* concept of .NET [eKFS06][1].

In the following sections, we will first introduce the concept of runtime-generated subclass proxies and its use for AOP, then describe our dynamic method-based extensions to the mechanism. We will evaluate the concept based on the classification scenario given in section 2.3.5 in chapter 2 and describe the join point and introduction model realizable with the approach. Concluding our analysis, we will give the results of a performance benchmark of existing implementations of the subclass proxy approach.

3.1 Subclass Proxies

A *proxy* P is defined to be an object which acts as a placeholder for a target object T [GHJV95]. Wherever T is expected, the proxy can be used instead,

[1] The contents of this chapter, as well as the classification scheme introduced earlier in chapter 2 mostly stem from the cited publication.

transparently extending the target object's behavior or controlling access to it without client code needing to be adapted. *Runtime proxies* are proxies created dynamically at runtime, without the programmer having to prepare a dedicated proxy class for every target class.

The idea of proxies extending target classes, adding new behavior without the target code needing to be extended, is similar to that of aspects extending the behavior of oblivious base code. This makes proxy approaches, and especially runtime proxy approaches, for which the system rather than the programmer generates the proxy code, a good candidate for an AOP weaving infrastructure.

The Microsoft .NET Common Language Runtime (CLR) already provides a runtime proxy mechanism called "transparent proxies" [Low03], which are needed for *.NET Remoting*—remote method invocation-based communication between different application domains, processes, and computers. Its uses as an AOP infrastructure are however limited by its functionality and its impact on application design.

Design-wise, it requires all target objects to be derived from a common base class: *System.ContextBoundObject*. This will not be an option in some cases, in other scenarios it might require unclean changes to the application design, which is contrary to the goals of AOP [KL+97].

Functionally, as it is designed for .NET Remoting, the transparent proxy mechanism can only extend behavior which occurs at the crossing of so-called "contexts", i.e. boundaries between application domains, processes, or computers. Although it is possible to create a single context for each target object, the transparent proxy mechanism cannot extend behavior of an object being accessed from its own context. This means that self calls—methods called on the *this* reference—cannot be extended. This would be a real drawback if the mechanism were used to realize a join point model. In addition, the static structure extension needed for the aspect-oriented feature of introduction can not be implemented using transparent proxies at all.

Positive aspects of the mechanism include that proxies are also *created* transparently because the CLR intercepts the instantiation of target objects and returns proxies instead. In addition, the CLR automatically corrects the *this* reference within T's methods—it refers to P instead of T, which is important if it is to be passed to other objects.

Since transparent proxies are suboptimal as an AOP infrastructure, alternative proxy mechanisms should be considered. In this process, it can be noted that the property of substitutability used previously for defining the term "proxy" is similar to the *Liskov Substitution Principle* (LSP) [LW94], which describes the relationship between *subtypes*. Like a proxy P, which can be substituted for an object T, the LSP states that an object of a subtype can be substituted for one of a supertype. This similarity can be used to implement proxies using the subtyping mechanisms present in .NET: interfaces and inheritance/subclassing. Figure 3.1 illustrates these proxy types, comparing them with transparent proxies and the unproxied scenario.

To realize a proxy using interface implementation—we call this an *interface proxy* approach—the target object T must implement a set of interfaces I and all client code must access T via these interfaces only. Then, a proxy object P

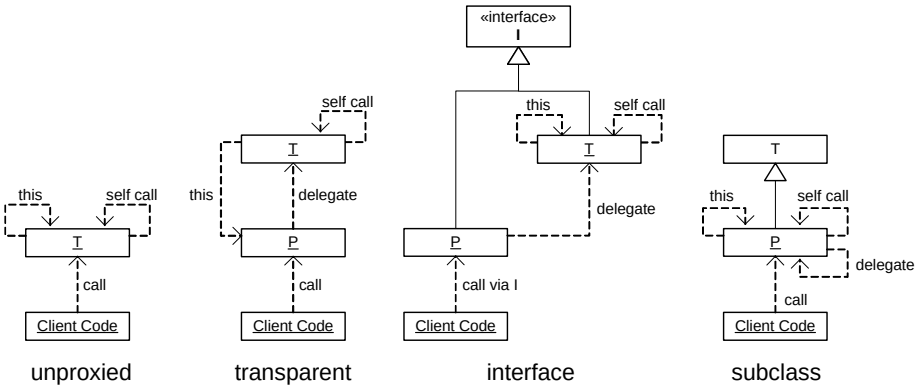

Figure 3.1: Proxy approach visualization

can be created which also implements I and holds a reference to T for delegation. T in the client code can be transparently replaced by P, which plays the role of a proxy. Within T's method implementations, however, the *this* reference refers to T rather than P, which is problematic if the reference is used to access the object: such access will not be registered by the proxy. In addition, like with transparent proxies, the interface proxy approach does not allow self calls to be extended, it would therefore be a suboptimal AOP infrastructure as well.

Realizing proxies using inheritance—the *subclass proxy* approach—is different from the aforementioned approaches. Whereas transparent and interface proxies have an object instance P replacing a target object instance T (and delegating), inheritance allows proxy and target to be one and the same object: a class P is derived from the target class T, overriding its methods and delegating to the original implementation. When P is instantiated, one object instance implements both P's and T's functionality.

Since subclasses are subtypes in .NET, the LSP applies and instances of P can be used wherever instances of T are expected. Subclass proxies intercept self-calls correctly, the *this* reference is automatically correct, and introduction is possible via interface implementation (see below).

In contrast to transparent proxies, both interface and subclass proxy have the disadvantage of needing a class factory [GHJV95] to make object creation transparent to client code. Table 3.1 summarizes the properties of the different proxy approaches, positive characteristics are shown in boldface. From this table, it is apparent that subclass proxies are the most appropriate proxy mechanism for creating an AOP infrastructure.

	Transparent	**Interface**	**Subclass**
Parent class	ContextBoundObject	**arbitrary**	**arbitrary**
Creation	**transparent**	factory	factory
Usage	**direct**	interfaces	**direct**
This reference	**P**	T	**P**
Extend self calls	no	no	**yes**
Introduction	no	**yes**	**yes**

Table 3.1: Properties of proxy approaches

3.1.1 Runtime Subclass Proxies

In the simplest form, subclasses do not implement a runtime proxy approach: the programer needs to write dedicated derived classes for each target type, manually overriding the methods that need to be extended. Using code generation, this can however be generically performed at runtime by a tool or framework. The .NET Base Class Library provides two powerful mechanisms allowing for runtime code generation: the *System.Reflection.Emit* namespace contains low-level classes and methods to dynamically generate .NET assemblies and types, *System.CodeDom* provides base classes for higher-level code generation. In this chapter, we will concentrate on *System.Reflection.Emit*.

Listing 3.1 shows how to dynamically generate a subclass of an arbitrary type at runtime using *System.Reflection.Emit*; this can for example be used to create a subclass proxy. The code first asks the current application domain to define a dynamic assembly, naming it "proxies", which in turn is used to define a dynamic module named "proxies" as well. By giving the module a DLL file name and using the *RunAndSave* flag when creating the assembly, it is possible to save the module to disk after generation in addition to using its types. The dynamic module is then used as a factory for the subclass type to be created. The type's base class is specified to be *baseType*, the parameter passed to the method, its access attribute is *Public* to make it publicly accessibly from other assemblies, and its name is defined by attaching "___Subclass" to the base type's name. By calling its *CreateType* method, the dynamically created subclass is finished and the corresponding *Type* object is returned and can be instantiated using *System.Activator.CreateInstance*.

```
public Type DefineSubclass(Type baseType) {
 AssemblyBuilder a = AppDomain.CurrentDomain.DefineDynamicAssembly(new
   AssemblyName("proxies"), AssemblyBuilderAccess.RunAndSave);
 ModuleBuilder m = a.DefineDynamicModule("proxies", "proxies.dll");
 TypeBuilder subtype = m.DefineType(baseType.Name + "___Subclass",
   TypeAttributes.Public, baseType);
 return subtype.CreateType();
}
```

Listing 3.1: Creating a subclass at runtime

3.1.2 Weaving Based on Subclass Proxies

Aspect-oriented programming is based on two main concepts: join points, i.e. points in the imperative program flow where aspects' advice methods are triggered, and introduction of new members to the aspects' target classes. Both concepts can—to a degree—be implemented with subclass proxies; the method of doing so is described in this section. An analysis on the join point model which is gained from this mechanism is performed later in section 3.3.1.

Join Points

By overriding the methods of its base class, a proxy class can provide replacement code for them, delegating to the original (base) implementation if necessary and executing additional code before, after, and instead of method executions. This effectively implements *before, after, and around method execution*

join points, with advice code being called (or even inlined) from the proxy's method overrides.

With *System.Reflection.Emit*, overriding methods is easily possible by inserting code prior to calling *TypeBuilder.CreateType*. Listing 3.2 shows how to override all virtual methods of the given base type. It does so by using the .NET *Reflection* mechanism to find all the public and nonpublic instance methods of the base type, checking whether they are virtual, and, if yes, defining a method with the same name, return type, and parameter types. The parameter types are found by inspecting the parameter information of the base method (the code uses *ConvertAll* and an anonymous delegate for brevity); the override's return type is the same as that of the base method.

```
foreach (MethodInfo m in baseType.GetMethods(
   BindingFlags.Public | BindingFlags.NonPublic |
   BindingFlags.Instance)) {
 if (m.IsVirtual) {
   ParameterInfo[] parameters = m.GetParameters();
   Type[] parameterTypes =
      Array.ConvertAll<ParameterInfo, Type>(parameters,
      delegate(ParameterInfo parameter)
      { return parameter.ParameterType; });
   MethodBuilder subMethod = subtype.DefineMethod(m.Name,
      MethodAttributes.Virtual | MethodAttributes.Public,
      m.CallingConvention, m.ReturnType, parameterTypes);
   ILGenerator il = subMethod.GetILGenerator();
   il.Emit(OpCodes.Ldarg_0);
   foreach (ParameterInfo parameter in parameters) {
     il.Emit(OpCodes.Ldarg, parameter.Position + 1);
   }
   il.EmitCall(OpCodes.Call, m, null);
   il.Emit(OpCodes.Ret);
 }
}
```

Listing 3.2: Overriding methods

The code snippet then defines the override's method body via IL (intermediate language) opcodes. The body loads the object reference (argument 0) and the parameters, calls the base method, and finally returns to the caller. An AOP approach can insert additional code into the body, implementing before, after, and around advice and delegating back to the original method if desired.

Although method join points are very important, they are not the only join point kind that should be supported by an AOP infrastructure. *Property get and set join points* are equivalent to method join points, since all properties are backed by respective getter and setter methods. *Construction and creation join points*, triggered when an object is (to be) instantiated, can easily be implemented by the factory used to create the proxy types and their instances. *Finalization join points*, triggered when the .NET garbage collector finds the lifetime of an object to be ended, can be implemented as a special kind of method join point by overriding the *Finalize* method of the object. Field get and set join points cannot however be implemented with subclass proxies.

Introduction

As opposed to static AOP approaches, runtime weaving approaches cannot simply introduce new members to a class. While it would be easily possible to add

these members to a subclass proxy, client code uses the proxy transparently and has no way of accessing the introduced entities with a statically typed programming language. The only form of introduction easily conceivable for runtime approaches is interface introduction: an aspect can add an interface and its implementation to an object, and client code can cast its object reference to the interface type. Since the proxy implements the interface, the cast succeeds.

Interface introduction can be easily implemented with *Reflection.Emit* by having the dynamically created subclass implement the interface. This is similar to listing 3.2 and therefore not separately demonstrated here.

Introduction via Mixins For reasons of flexibility, the interface implementation within the proxy subclass should be able to delegate to a separate implementation of the introduced interface, a so-called *mixin*. The mixin mechanism, which is similar in concept to the notion of interface proxies, allows aspects to define an interface implementation in their own context and pass it to the proxy, where it gets invoked when client code accesses the proxy via the introduced interface. This flexibility makes for a very powerful means of separation of concerns. Figure 3.2 demonstrates the class layout of mixins in combination with subclass proxies.

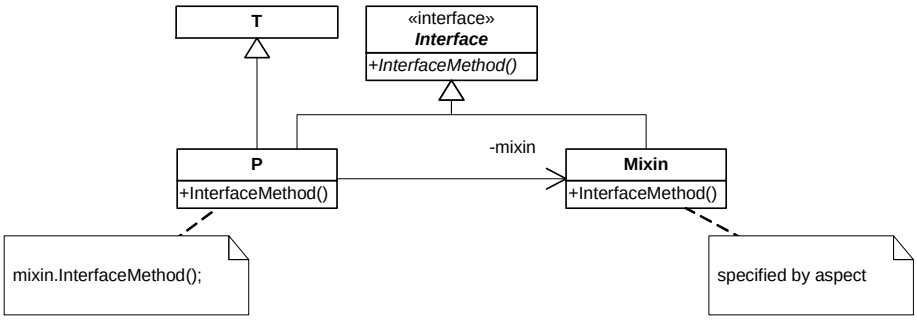

Figure 3.2: Class layout of mixins with subclass proxies

3.2 Privileged Access to Target Objects' Internals

One important property of aspects is that they often require more privileged access to their target object's internals than other objects should have. This is because they implement cross-cutting concerns, whose implementations can be tightly coupled to the objects they cut. While a subclass proxy naturally has access to all public and protected (family-accessible) fields and methods of its base class, it has no access to private or assembly-visible members.

.NET provides a reflection mechanism to work around this: given the necessary rights, every object can reflect over another object's private fields and methods in order to inspect and change the fields' values or invoke the methods. However,

.NET Reflection is not optimized for performance: our tests have shown that accessing a field via reflection is around 200 to 700 times slower than direct access, and still 180 times slower than invoking an accessor method would be. Since field access is such a basic operation, this might conceivably slow down an aspect-oriented application, depending on the degree of coupling between aspects and object state.

Since direct field access cannot be implemented from a subclass and reflective access is so slow, it would be desirable to at least have accessor methods for those private fields required by an aspect. Unfortunately, such a method cannot be added to a subclass, which has no access to private members. As a solution, with .NET 2.0 there is a new mechanism called *Lightweight Code Generation* (LCG), or *Dynamic Methods* [Mic06a]. It allows methods to be generated at runtime which can be attached to any existing type, allowing access to all its private data. Access to the method is provided via a delegate, allowing flexible invocation which is still 15 to 20 times faster than reflection-based field access. The diagram in figure 3.3 shows a performance comparison of the different ways of accessing fields (measured on an Athlon XP1800+ with 512 MB RAM and .NET framework v2.0.50727).

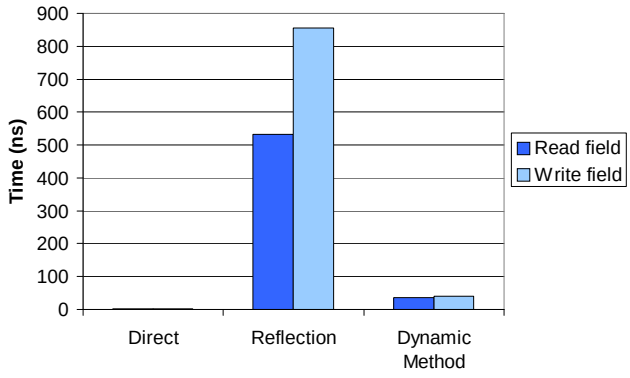

Figure 3.3: Access method performance

3.2.1 Field Access Framework

For an AOP approach based on subclass proxies, we suggest a field access infrastructure, as illustrated in listing 3.3. It consists of strongly typed wrappers *Setter* and *Getter* for accessor methods, an accessor method generator *MethodGenerator*, which generates the methods using LCG, and a wrapper structure for fields, which automatically initiates the accessor method generation when being constructed and provides a *Value* property delegating to the accessor methods for convenient use.

The listing shows the source code for these infrastructure entities. The method body constructed by *CreateSetter* simply loads the given target object (argument 0) followed by the value (argument 1), which is then stored in the given field before returning. The method body constructed by *CreateGetter* first loads

the target, then loads the field value, and then returns, leaving the field value on the evaluation stack for it to be returned to the caller. Because the created dynamic methods are associated with the target type (*ClassType*), they can safely access even private fields using the *ldfld* and *stfld* opcodes.

```
delegate FieldType Getter<ClassType, FieldType>(
   ClassType target);
delegate void Setter<ClassType, FieldType>(
   ClassType target, FieldType value);

class MethodGenerator {
 public static Setter<ClassType, FieldType> CreateSetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedSetter", typeof(void),
     new Type[] { typeof(ClassType), typeof(FieldType) },
     typeof(ClassType));
  ILGenerator ilGenerator = newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldarg_1);
  ilGenerator.Emit(OpCodes.Stfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Setter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Setter<ClassType, FieldType>));
 }

 public static Getter<ClassType, FieldType> CreateGetter
    <ClassType, FieldType>(FieldInfo fieldInfo) {
  DynamicMethod newMethod = new DynamicMethod(
     fieldInfo.Name + "___GeneratedGetter",
     typeof(FieldType), new Type[] { typeof(ClassType) },
     typeof(ClassType));
  ILGenerator ilGenerator = newMethod.GetILGenerator();
  ilGenerator.Emit(OpCodes.Ldarg_0);
  ilGenerator.Emit(OpCodes.Ldfld, fieldInfo);
  ilGenerator.Emit(OpCodes.Ret);
  return (Getter<ClassType, FieldType>)
     newMethod.CreateDelegate(
     typeof(Getter<ClassType, FieldType>));
 }
}

struct Field<ClassType, FieldType> {
 public readonly FieldInfo FieldInfo;
 public readonly ClassType Target;
 public readonly Getter<ClassType, FieldType> Getter;
 public readonly Setter<ClassType, FieldType> Setter;

 public Field(ClassType target, FieldInfo fieldInfo) {
  this.FieldInfo = fieldInfo;
  this.Target = target;
  this.Getter = MethodGenerator.CreateGetter
     <ClassType, FieldType>(fieldInfo);
  this.Setter = MethodGenerator.CreateSetter
     <ClassType, FieldType>(fieldInfo);
 }

 public FieldType Value {
  get { return Getter(Target); }
  set { Setter(Target, value); }
 }
}
```

Listing 3.3: Infrastructure for efficient field access

3.2.2 Other Uses of LCG for AOP

In the context of AOP, dynamic methods cannot only be used for accessing private fields (and methods), they also constitute an easy way of adapting an already generated subclass proxy to new requirements. Consider a proxy extending a target object with an advice method that takes join point context information. In order to correctly pass the information needed by the advice, a subclass proxy approach must either settle for untyped advice parameters (e.g. an argument of type *object[]* which is filled with the parameters) or generate adapter code. Such adapter code generation can only be done at the time of proxy generation. Therefore, if a new aspect is to be attached to an instance of a previously generated proxy class, adapter code may not be available, thus restricting the runtime adaptability facilities of a subclass proxy infrastructure.

With dynamic methods, however, it is possible to generate adapter code even after proxy generation has finished and thus react to changing aspect bindings. This makes subclass proxy weaving truly dynamic, enabling aspect reconfiguration to be performed at any time at runtime (not only before object instantiation). Figure 3.4 illustrates the working of adapter code: without an adapter, the proxy class P needs to directly bind its method M to the aspect's advice method (requiring the s parameter to be passed through to the aspect). With an adapter, P can pass all its arguments to the adapter method which then selects the arguments to be passed to the aspect. With the latter mechanism, an aspect can be added at runtime without P needing to change.

(Note that, while runtime generation of whole adapter *types* was already possible without LCG, this had a much higher memory footprint. In addition, the garbage collector cannot remove unneeded types from memory, but it can remove dynamic methods when they aren't needed any longer [Mic06b].)

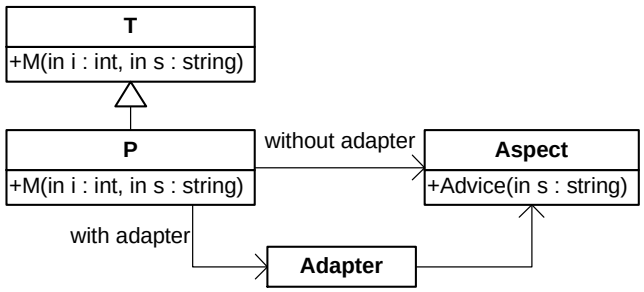

Figure 3.4: Binding to join points with and without adapter

3.3 Conceptual Analysis

Classified by the schema given in section 2.3.5, runtime-generated subclass proxies make for an augmenting approach performed at runtime, thus resulting in beneficial properties for invasiveness, debuggability, implementation effort, compatibility, and language support. It has negative properties regarding the join

point model (to be discussed in detail), design prerequisites (a factory is needed for creating proxied instances), and performance (also to be detailed).

3.3.1 Join Point Model

From an AOP perspective, a number of join points can be implemented using subclass proxies, whereas others can't. Table 3.2 characterizes the join point model realizable with the approach. Using a source code weaving tool, all the join points shown could be realized.

Join Point Type	Before	Instead of	After
Object creation	**yes**	**yes**	**yes**
Constructor execution	**yes**	no	**yes**
Class construction	no	no	no
Object finalization	**yes**	**yes**	**yes**
Method execution	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Method call	no	no	no
Property get	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Property set	**yes** (virtual)	**yes** (virtual)	**yes** (virtual)
Field get	no	no	no
Field set	no	no	no
Exception thrown	no	no	no
Exception caught	no	no	no
Exception escaping	**yes** (virtual)	**yes** (virtual)	-
Construct (for, if, ...)	no	no	no

Table 3.2: Join point model with subclass proxies

While this join point model is definitely restricted when compared to that of a source code weaving tool, we believe that this is not a problem in most AOP scenarios. When an application is designed from scratch in an aspect-oriented way, all join points are known in advance, before any of the classes or aspects are to be implemented. With a subclass proxy approach, the design would naturally evolve around the join point kinds being available, ignoring those which can't be used. In most cases, however, small design changes can work around the missing join point types.

For example, because field access join points cannot be realized using subclass proxies, a design guideline could be created to access fields via accessor methods (or properties) only, which is a common guideline with object-oriented programming anyway. Those methods needed as join points would be defined to be virtual. The only join points which can't be worked around are: instead-of constructor execution, class construction, exceptions thrown and caught in the same method, and join points at a statement-level granularity. In addition, subclass proxies cannot advise non-virtual methods or distinguish between method call and execution.

3.3.2 Psychological Factors

Adoption of AOP is hindered by many factors, which are remedied to a great extent by the use of an approach based on subclass proxies:

AOP can make a program incomprehensible: AOP as an invasive mechanism is often regarded with distrust, because aspect-oriented tools weave code

together as a black box. Since aspect definitions aren't necessarily visible when reading base source code, it is hard to make conclusions about the actual runtime behavior of the code. Similar to the object-oriented mechanism of dynamic dispatch, but in a more poweful and more extensive way, the code being executed is not necessarily the code which can be inspected in a class definition. With this prerequisite, concerns about reliability and debuggability (if an error occurs in mangled code, will it be retraceable to the original source code?) as well as the question of unpredictable execution paths in woven code naturally arise with invasive tools. In contrast, subclass proxies are built on established object-oriented concepts such as method overriding and interface implementation. These are well-known, don't introduce reliability or debuggability problems, and developers can comprehend what happens at runtime.

Adaption to new tools: Aspect-oriented tools often replace the tools (e.g. compilers) developers are used to instead of augmenting them. With all approaches, developers need to adapt to new tools with new error messages, longer or different update cycles, and sometimes incompatibilities with the original tools. Since subclass proxies can be implemented as a framework or class library, there is no need to switch tools with such an approach—developers can continue using their familiar environment and still obtain the benefits of AOP.

Unfinished tools: AOP tools usually need a lot of work, this is the cause of the lack of production quality .NET-based AOP tools. However, since subclass proxies are much simpler to implement than code weaving tools, the probability of reaching production status is much higher with this approach.

AOP based on subclass proxies therefore has high adoption potential. With the described prerequisites, users should be easily convincable of the new technology.

3.4 Performance Evaluation

With proxy-based approaches, aspect code is not directly inserted into the target code; object-oriented mechanisms are used instead. This is often regarded as a performance disadvantage of such approaches. On the .NET platform, however, most optimizations are not done by a language compiler inlining code, but by the JIT compiler's optimizer at runtime. There are some restrictions to JIT optimization with subclass proxies, because virtual method calls to the target object are always performed through the proxy and can't be replaced by ordinary calls, but these apply just as well when the application makes use of the object-oriented mechanisms itself. Most JIT optimizations should not be affected adversely by the use of subclass proxies.

In this section, we will take a look at two implementations of the subclass proxy mechanism—NAspect and DynamicProxy [Ver04]—and analyze object construction time and method call time, since these represent the main points during program flow where a proxy-based mechanism performs differently from a mechanism based on code weaving.

3.4.1 Object Creation

The first time an object is created from a target type, the proxy-creating factory must construct the new proxy subclass. This is a lengthy operation, our measurings have shown this to take up to 37ms (DynamicProxy) and 12ms (NAspect), as opposed to the few nanoseconds an ordinary new operation (usually) needs. Seen as isolated numbers, this is a tremendous slowdown.

However, analysis of cross-cutting concerns in space-based computing [eKS05a] reveals common scenarios which only have few types being aspectized at the same time, with a higher number of instances created from those. In such scenarios, the generated proxy subclasses can and should be cached, making an instantiation consist of one hashtable lookup plus one call to the type's constructor (either via Reflection or, optimized, via a delegate), which takes a few hundred microseconds at most in our measurements. In the use cases we studied, this makes instantiation time of proxied objects not a problem. On the other hand, if it is vital that proxied objects of many different types are created with rigid performance requirements (a few nanoseconds per instantiation), pure proxying might not be the mechanism of choice; pooling and flyweight techniques [GHJV95] can improve on that.

Figure 3.5 shows the benchmark of an instantiation benchmark done with 1000 different types on an Intel Pentium M4 1.8 GHz with 512 MB RAM.

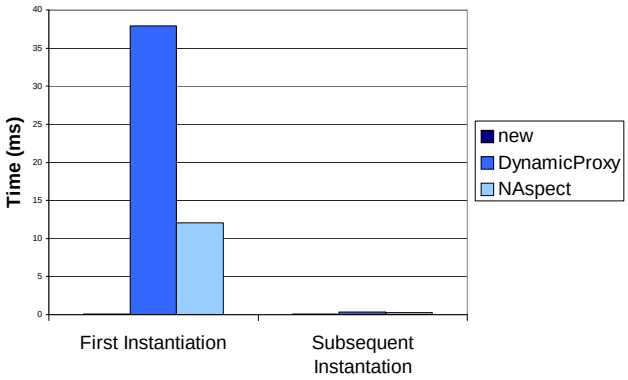

Figure 3.5: Instantiation benchmarks

Regarding memory usage, an AOP tool based on subclass proxies should use as few dynamic assemblies and modules as possible. Our tests have shown this to scale much better than having one assembly per proxied type. Caching of the generated proxy types will also improve memory footprint. A user should be aware that the only way to remove the generated proxy types from memory is by unloading their application domain (of course, their *instances* are garbage collected as usual), although again this will not be an issue in scenarios with a reasonable number of aspectized types.

3.4.2 Method Invocation

Method join point performance is more important than object creation performance because—as can be seen for example by analyzing the classes in .NET's Base Class Library—the frequency of method calls as compared to object instantiations is typically very high. With subclass proxies, method join points are implemented via method overrides. A method join point of an optimal proxy is therefore no different from a virtual method call (a few nanoseconds) plus one non-virtual base call if delegation to the original code is needed (a few nanoseconds as well). This optimal approach however requires injection of advice code into the subclass proxy, which is not trivial to implement. Current implementations therefore choose not to directly invoke the base method from within the override. Instead, they encapsulate the base call and hand it to an interceptor provided by the aspect, which may then choose to invoke the method or not. For this encapsulation, DynamicProxy constructs a delegate, whereas NAspect relies on Reflection. Both approaches are not ideal what regards method interception performance, although delegates are an order of magnitude faster than Reflection.

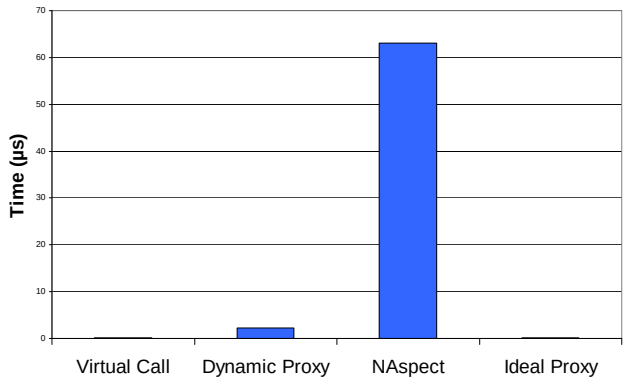

Figure 3.6: Method call benchmarks

Figure 3.6 shows a method call benchmark done with an AMD XP1800+ system. The values for ordinary virtual call, DynamicProxy, and NAspect are measured, the value for the ideal proxy is calculated—an implementation can achieve this performance if call times are of much importance. We measured the call and return time of empty methods (with the proxies delegating to the original empty methods); in real scenarios, these values have to be seen in relation to concrete method execution time. For example, our tests have shown that with an average method whose body needs several microseconds for execution, the measured call times are not that significant. This might well be the reason why existing subclass proxy implementations have not yet chosen to implement the ideal approach.

To summarize, while current implementations show medium to high method call slowdowns, an ideal subclass proxy approach can lead to call times in the range of nanoseconds, not much higher than ordinary method calls. Even the call times of current implementations are less significant if the called methods

have nontrivial bodies.

3.5 Concluding Remarks to Subclass Proxies

In the previous sections, we have motivated, described, and analyzed the subclass proxy mechanism as an implementation infrastructure for aspect-oriented programming. We compared the different proxy approaches available on the .NET platform, identifying subclass proxying to be the most powerful one. Classifying the weaving approach implementable with subclass proxies, we have shown the disadvantages of the model, such as a more constrained join point model and design restrictions, but have also identified technical advantages over classical implementation mechanisms, such as easy debuggability and runtime weaving capabilities.

Performance benchmarks have shown current subclass proxy implementations to be of mediocre performance with regards to intercepted method calls. However the proxy concept could be improved in this regard in order to achieve call times not much different from ordinary virtual method calls if necessary.

Analyzing the psychological properties of subclass proxies, we have identified a high adoptability potential of the non-invasive mechanism which requires no dedicated compiler tools. With such prerequisites, industrial acceptance of an aspect-oriented programming tool based on subclass proxies should be possible.

Chapter 4

XL-AOF

After having introduced an infrastructure for aspect-oriented weaving on the .NET platform, the next step towards an efficient space-based programming experience is XL-AOF, an *extensible* and *lightweight* framework for aspect-oriented programming on the .NET platform. It is designed especially for the development of a declarative and well-encapsulated interface for space-based network abstractions, but can be used for general aspect-oriented programming as well.

Before introducing the framework, we will first define the notions of *light weight* and *extensibility*, as these are the two fundamental requirements for the aspect-oriented framework.

4.1 Light Weight, Extensibility, and Adoptability

Following the definition Matthew Deiters gives in his article "Aspect-Oriented Programming" [Dei05], we denote approaches to be *lightweight* implementations of the aspect-oriented paradigm if they have minimal impact on the system as a whole. An approach (or tool) is of light weight if

- It can be cleanly integrated into the tool chain (IDE, debugger, compiler, disassembler, etc.) used by the programmers building a system,
- It is naturally integrated into the mainstream programming languages the system is implemented in,
- It is noninvasive, i.e. it extends the base code without modifying it, and
- The target program can decide at runtime whether to use it or not.

A lightweight approach usually provides less freedom in the development of an aspect-oriented tool: due to the property of noninvasiveness, a tool has to rely on other mechanisms than code instrumentation to invoke aspect code at join points, and this usually reduces the set of join points the tool can offer to the programmer.

We define an approach or tool to be (easily) extensible if new modularized cross-cutting concerns can easily be added even though they were not considered at the tool's design time. This is in fact a property of most current AOP tools, but it clearly distinguishes these approaches from application containers providing a fixed set of services (implementing cross-cutting concerns) to the programmer.

We define it to be (easily) adoptable if

- Its aspect language is easy to learn and use,
- It does not require complicated and time-consuming build or application configuration,
- It relies on well-known and easily understandable infrastructural concepts,
- It does not need complicated or hard-to-use build or weaving tools, and
- It does not break any mainstream tools typically used for building the system.

Since light weight and adoptability have similar requirements, they go hand-in-hand, and lightweight implementations are typically easier to adopt than heavyweight ones.

As already indicated in chapter 3, we base XL-AOF on the runtime-generated subclass proxy mechanism (we use the *DynamicProxy* implementation [Ver04]), which, being a highly compatible, noninvasive, runtime-based, and framework-implemented infrastructure, fortunately does not stand in the way of fulfilling the requirements stated above. It is, of course, far too low-level to be considered easily adoptable; achieving this will be the task of XL-AOF. While describing the aspect features of XL-AOF, we will therefore analyze each feature's impact on adoption.

4.2 *ObjectFactory* as an Entry Point

As already indicated in chapter 3, tools based on runtime-generated subclass proxies must provide a factory, which has to be used for target object instantiation instead of the ordinary *new* operation. In the factory, the AOP tool analyzes the target object's class, finds all applying aspect bindings, and instructs the infrastructure to weave (i.e. produce a subclass proxy) accordingly.

With XL-AOF, this task is performed by a singleton class *ObjectFactory*, which provides a generic factory method *Create*<*T*> for type-safe creation of objects of arbitrary types, as shown in listing 4.1.

```
public static class ObjectFactory {
  public static T Create<T>(params object[] constructorArgs) { ... }
  public static object Create(Type aspectType, object[] constructorArgs);
  ... // additional methods not shown here
}
```

Listing 4.1: *ObjectFactory* class

The *Create*<*T*> method takes a variable parameter list of arguments which are to be passed to the constructor. The non-generic variant of the *Create* method is equivalent, but takes the target type as a reflected *Type* object rather than a type parameter, which is better-suited for reflective scenarios. The object factory also has additional methods for aspect configuration and custom join point handling, which will be explained in section 4.3.2 and 4.3.3.

For a user of XL-AOF, the *ObjectFactory* class is the entry point to aspect-orientation. Objects created with the *Create*<*T*> method are guaranteed to be at least of type *T*, but if an aspect applies to them, they will actually be of a derived proxy type. For the weaving to work as expected, a target class must be extensible, i.e. it must not be sealed or inaccessible.

Positive Adoptability Issues

- *ObjectFactory* provides a simple central entry point to aspect-orientation.
- For the user, there is no need to be concerned about the underlying weaver and infrastructure, there is also no need to pass any aspect context information to the factory.
- The intention and external working model of the factory is easy to understand.
- One factory can be used for all object instantiations, even if the object does not have any aspects associated with it.
- The *Create*<*T*> method is generic, thus allowing objects of arbitrary types to be created without typecast.

Negative Adoptability Issues

- The *Create*<*T*> method is not statically safe: it allows an argument list to be specified which has no corresponding constructor. Such an error can only be found at runtime.

  Comment Unfortunately, this cannot be helped with .NET's current generics support if a general object factory is needed. The only (cumbersome) workaround is to build concrete factory methods for each type needing static safety. These factory methods would need to have argument lists corresponding to the relevant constructors and would delegate to the *ObjectFactory*.

- Object instances created with the standard *new* operation will not be aspectized.

  Comment As .NET currently does not allow interception of the *new* operation, this cannot be resolved.

- Classes must be extensible, i.e. not be sealed or inaccessible.

  Comment As we use a non-invasive, augmenting infrastructure, this is indeed an essential demand.

4.3 XL-AOF Declarative Aspect Language

Although XL-AOF is based on a framework approach and does not define a new aspect-oriented programming language, it provides an aspect-oriented feature set, mostly implemented in the form of declarative attributes. We call this feature set, which is one of the distinguishing and novel characteristics of our work, the *aspect language* of XL-AOF, and we will describe it in the following sections.

4.3.1 Aspect Definition

Like other aspect-oriented toolkits, XL-AOF provides a mechanism for encapsulating cross-cutting concerns—the aspect. However, similar to only a few other tools, XL-AOF does not define a new modularizing language concept for this purpose, but instead reuses the existing object-oriented modularizing entity: the class.

In XL-AOF, an aspect is simply defined as a class. There are no special rules or constraints for aspect classes, an aspect can have fields, methods, and constructors just like any other class, and it can also be instantiated just like any other class. It becomes an aspect (with pointcuts, advice, and introduction elements) only by aspect configuration (see below, section 4.3.2), and can be a target class of other aspects just as well.

Positive Adoptability Issues

- Aspects are declared as ordinary classes, which does not require any new concepts or language mechanisms to be learned.
- There are no constraints for aspect classes, they can be derived from any base class and implement any interfaces.
- Aspects can be instantiated by user code as well as by XL-AOF, which allows their functionality to be used and, most important, tested independently of their target classes.

4.3.2 Aspect Configuration

In order to write an aspect-oriented application, it is necessary to specify what aspects should be bound to what target classes. With XL-AOF, this is even more important, since aspects are defined as ordinary classes and become aspects only by being configured to be bound to another class. In fact, aspects can also be bound to other aspect classes, allowing *aspects of aspects* to be realized.

XL-AOF provides three different ways of binding aspects to target classes, which are explained in the following sections with ascending flexibility, but also with increasing complexity. All of them are, however, based on the concept of *aspect factory attributes*. Factory attributes are .NET custom attributes used to specify the binding, which at the same time provide functionality for actually

instantiating the aspect. An aspect factory attribute is defined as an ordinary custom attribute class, which has one factory method tagged with an *AspectFactoryAttribute*.

As an example, listing 4.2 shows the declaration of a factory attribute, whose factory method creates an instance of an exemplary *LogAspect*. When this attribute is involved in the aspect configuration of a class which is instantiated via the object factory, its factory method will be called in order to instantiate the respective aspect. Factory methods must either have an empty argument list or (as in listing 4.2) take the target type and the constructor arguments of the instance as parameters.

```
class LoggedAttribute : Attribute {
  [AspectFactory]
  public LogAspect Create(Type aspectizedType, object[] constructorArgs) {
    return ObjectFactory.Create<LogAspect>();
  }
}
```

Listing 4.2: Factory attribute definition for an exemplary *LogAspect*

Because the factory attributes can decide when to create a new object and when to return an existing (cached) one, they implement the aspect scopes known from other AOP approaches (AspectJ [tAT05], AspectWerkz [Bon04]): if they return a new instance for every invocation (as in listing 4.2), one dedicated aspect instance is created for every constructed object instance, effectively implementing the binding mechanism known as *per object*, *per instance*, or *per target* scope. If a factory attribute class always returns the same aspect instance, caching it between calls to its factory methods, it implements a singleton mechanism also known as *per VM* scope. A factory attribute can also implement more sophisticated caching mechanisms, for example instantiating one aspect per aspectized type (*per class* scope) or one per thread (*per thread* scope). Currently, XL-AOF does however not support *per joinpoint* scopes, where a new aspect instance is created for every joinpoint reached at application runtime.

Discussion of Novelty The idea of having factory attributes for aspect instantiation is new; to our knowledge no other AOP tool exposes so much control over how an aspect is instantiated. The approach makes aspect creation much more powerful than with the usual automatic scope management mechanisms, allowing for many kinds of user-defined instantiation schemata. For example, the aspect's constructor arguments could be taken into account for instantiation, e.g. for implementing some kind of multi-singleton, where two equal combinations of parameter values always cause the same cached aspect instance to be returned. Instances could also be taken nondeterministically from a pool of aspects to implement some kind of aspect load balancing. And instantiation can also be declaratively configurable by adding a set of attribute arguments to the factory attribute.

Of course, the mechanism by itself is also more complicated than predefined scope-based aspect creation. Therefore, XL-AOF already contains a number of predefined factory attributes for the most common configuration purposes (see below). This makes the attribute-based approach as easy to use as a predefined one would be, but still leaves all the power for situation where it's needed.

Positive Adoptability Issues

- Factory attributes can implement different (and novel) instantiation scenarios and scopes.
- Aspect classes can be target classes of other aspects, allowing aspects of aspects to be realized.

Negative Adoptability Issues

- Factory methods are imperative and make the attribute-based approach more complicated than comparable built-in scopes as they are provided by other AOP tools.

  Comment There are easily usable attributes predefined for most simple scenarios. New attributes only have to be defined for aspects requiring complex initialization logic or for novel instantiation scenarios, which are use cases not supported by other tools. Also, having dedicated attributes for reusable aspects delivered in a library can constitute a means of in-code documentation of the aspects' effects.

Declarative Type-Level Configuration

The simplest way to bind an aspect to a target type is to apply the aspect's factory attribute to the target class definition. For example, listing 4.3 defines an exemplary target class *Account* and associates it with the *LogAspect* from listing 4.2 by annotating it with the *LoggedAttribute* factory attribute.

```
[Logged]
public class Account {
  ... // methods not shown
}
```

Listing 4.3: Aspect configuration at type level

This way of configuring aspects has the advantage of good understandability and self-documenting code: by looking at a class definition, it is immediately clear what cross-cutting concerns influence the class, even though these are not tangled within the class. It is, however, contradictory to the obliviousness property given in section 2.3.2 in chapter 2—when a class declaration (indirectly) enumerates the applying cross-cutting concerns, it is by definition not oblivious of these concerns, and an approach relying on such an enumeration is not aspect-oriented according to the definition of Filman and Friedman [FF00]. As also stated in that section, this definition is, however, heavily disputed, and we ourselves believe that in many cases, increased understandability and self-documentation is more important than target code obliviousness.

For this aspect configuration mechanism, XL-AOF provides three predefined factory attributes: *PerObjectAspectAttribute*, *PerClassAspectAttribute*, and *SingletonAspectAttribute* are general implementations for aspects without complicated initialization logic and can readily be used for configuring any instance-, class-, and singleton-scoped aspects. They take the type of aspect to be bound

to the respective target class as an attribute parameter and also allow additional (optional) attribute parameters to be passed to the aspect's constructor. Listing 4.4 illustrates the use of the *PerObjectAspectAttribute*, again by applying the exemplary *LogAspect* to the *Account* class of listing 4.3, this time however by employing a standard factory attribute. The predefined attributes clearly communicate aspect scope and make the definition of custom factory attributes unnecessary for most aspects.

```
[PerObjectAspect(typeof(LogAspect))]
public class Account {
  ... // methods not shown
}
```

Listing 4.4: Aspect configuration at type level with a standard attribute

PerClassAspectAttribute and *SingletonAspectAttribute* are employed in the same way as *PerObjectAspectAttribute* and only differ in aspect scope and instantiation model.

Positive Adoptability Issues

- Factory attributes applied to target classes are an easily learned mechanism, which makes for self-documenting and understandable code.
- The adoptability concern of feared incomprehensibility (see section 3.3.2 in chapter 2), which is already addressed by employing subclass proxies as an infrastructure, is further mitigated by attaching aspect bindings to the target classes.
- A number of easily-understandable predefined attributes are available for the most important configuration scenarios.

Negative Adoptability Issues

- Attaching aspect-configuration to target class definition violates the principle of target code obliviousness and can therefore be seen not to fulfill the definition of AOP and to reintroduce cross-cutting, as configuration clauses would be required on all (otherwise unrelated) target classes of an aspect.

  Comment The definition requiring obliviousness is heavily disputed, and we believe such a configuration mechanism to be well-suited for many usage scenarios, especially for a declarative space-based programming language interface. What regards the problem of reintroducing cross-cutting code, we argue that the code introduced by aspect configuration is minimal when compared to fully tangled cross-cutting concerns. Many examples of this will be given in chapter 5. For situations, where obliviousness is absolutely required, two additional configuration mechanisms will be presented in the following sections.

Declarative Assembly-Level Configuration

While the simple type-level configuration approach is well-suited for many situations, use cases are conceivable where *oblivious* target classes are desirable or even necessary. For example, if the same application should be compiled with different aspect configurations, it would be desirable to have all configuration clauses in one place rather than on different class definitions. If a class is to be part of a class library, aspect configuration might depend on the actual application where the class is used. And generally, if an aspect configuration should be changed, but the target class cannot be recompiled, the type-level configuration approach isn't applicable either.

For these scenarios, another configuration mechanism is needed. While it might be tempting to implement a configuration mechanism at aspect level (where the aspect declaration specifies what target classes it should be bound to, as implemented by AspectJ [tAT03]), such aspects would not be reusable without recompilation, and the change would only relocate the problems.

A more sophisticated mechanism should separate aspect configuration from both target and aspect class declarations, allowing the aspect configuration to be changed without recompilation of either target or aspect code, in fact without requiring the source code of either to be available. XL-AOF supports such a mechanism by implementing a declarative assembly-level configuration mechanism: custom attributes of type *GlobalAspectBinding* and derived can be applied to any assembly linked with the application, allowing very flexible, yet still declarative aspect configuration.

Listing 4.5 shows the interface of the *GlobalAspectBindingAttribute*, which takes the target type and the factory attribute's type and arguments as attribute parameters, as well as an example configuration clause. The example again configures the *LogAspect* of listing 4.3.2 to be applied to an *Account* target class via the *LoggedAttribute* factory attribute. This configuration is semantically equivalent to listing 4.3, but it can be applied to any assembly linked to the application, which has the benefit of greater flexibility, but also the drawback of somewhat decreased code clarity.

```
[AttributeUsage(AttributeTargets.Assembly, AllowMultiple = true)]
public class GlobalAspectBindingAttribute : Attribute {
  public GlobalAspectBindingAttribute(Type targetType, Type attributeType, params
    object[] attributeArgs) {...}
  ... // implementation details not shown
}

// example configuration
[assembly:GlobalAspectBinding(typeof(Account), typeof(LoggedAttribute))]
```

Listing 4.5: Aspect configuration at assembly level

For assembly-level configuration, there also exist a number of predefined configuration attributes, which are the direct counterparts of the predefined standard factory attributes: *GlobalPerObjectAspectAttribute*, *GlobalPerClassAspectAttribute*, and *GlobalSingletonAspectAttribute*. Listing 4.6 shows an example configuration clause equivalent to that of listing 4.5, but using a predefined assembly-level configuration attribute.

```
[assembly:GlobalPerObjectAspect(typeof(Account), typeof(LogAspect))]
```

Listing 4.6: Aspect configuration at assembly level with a standard attribute

Positive Adoptability Issues

- Assembly-level attributes allow flexible, yet declarative aspect configuration without binding the configuration clause to either target or aspect class.
- Assembly-level configuration can be inserted in any assembly linked to the application, allowing reconfiguration without recompilation of either target or aspect class.
- All configuration attributes can be collected in a single place, allowing easy aspect reconfiguration, for example for building different editions of an application.

Negative Adoptability Issues

- Since assembly-level attributes can be attached to any assembly in the application, it is hard to see at a glance what aspects are applied to a class.

  Comment It is true that detached configuration makes code less self-documenting and less understandable. The assembly-level configuration mechanism should therefore only be used after careful consideration, and the configuration clauses should be collected in one or a few dedicated code files, making it easier to analyze an application. In the future, tool/IDE support could remedy the situation by providing visual clues as of what aspects apply to a target class.

Dynamic Configuration

While declarative assembly-level configuration is already very powerful, it still has some restrictions. Most notably, changes in aspect configuration require recompilation of the assembly containing the configuration clause, making it hard to add or remove aspects to/from a running application, for example as a means for patching or extending a long-running applications.

To overcome these limitations, XL-AOF provides an imperative API for changing an application's aspect configuration by attaching and detaching aspects to and from target types at any point during application runtime. As indicated in section 2.3 in chapter 2, such runtime weaving is a feature of dynamic AOP, which provides great flexibility, but often comes with only loosely defined semantics, because aspect reconfiguration could happen asynchronously to aspect execution. XL-AOF solves the semantical problem by defining a strict model for dynamic weaving (which is also enforced by its subclass proxy infrastructure): runtime changes in an application's aspect configuration always influence

only those target object instances which are created *after* the configuration has changed. All instances created prior to the change remain uninfluenced, and the *ObjectFactory* class ensures that instantiation and configuration changes cannot occur in parallel by sequentializing multithreaded access.

The dynamic aspect configuration API is implemented by two generic methods of *ObjectFactory*: *AttachAspect<TTarget>* and *DetachAspect<TTarget>*. Both take the target type of the aspect as type arguments and an instance of an aspect factory attribute as a parameter, as shown in listing 4.7. (Similar to the *ObjectFactory's* creational methods, both methods also exist in non-generic variants, which are easier to use in reflective scenarios.)

```
public static class ObjectFactory {
  public static void AttachAspect<TTarget>(object factoryAttribute) { ... }
  public static void AttachAspect(Type targetType, object factoryAttribute) { ...
    }
  public static bool DetachAspect<TTarget>(object factoryAttribute) { ... }
  public static bool DetachAspect(Type targetType, object factoryAttribute) {... }

  ... // additional methods not shown
}
```

Listing 4.7: Imperative API for dynamic aspect reconfiguration

Listing 4.8 shows a usage sample for the dynamic reconfiguration API, which again is equivalent to listing 4.3 in that it attaches the exemplary *LogAspect* to an *Account* class. It differs however in that the binding is performed at runtime and temporarily only: the aspect binding is removed again after an instance of the class is created, so the aspect configuration only influences this single *Account* instance. Although the example uses the custom *LoggedAttribute*, any of the predefined factory attribute described in section 4.3.2 can equally well be used with the imperative approach.

```
public Account CreateLoggedAccount() {
  object factoryAttribute = new LoggedAttribute()
  ObjectFactory.AttachAspect<Account>(factoryAttribute);
  try {
    return ObjectFactory.Create<Account>();
  }
  finally {
    ObjectFactory.RemoveAspect<Account>(factoryAttribute);
  }
}
```

Listing 4.8: Dynamic aspect reconfiguration

For removal of an aspect from the configuration, the exact attribute instance which was used for registration is needed. If this instance is not available, for example because the aspect was configured declaratively, aspect introspection (see section 4.3.6) must be used to obtain it.

Although dynamic runtime reconfiguration is the most powerful aspect configuration mechanism, it should only be used if absolutely necessary (e.g. with applications which cannot be restarted). If used carelessly, it can lead to complicated programs, which are very difficult to analyze and understand. Also, the task of aspect configuration can in itself be seen as a cross-cutting concern, so tangling and scattering should sought to be avoided.

Positive Adoptability Issues

- Runtime aspect reconfiguration provides a means for dynamic AOP, giving much flexibility, but within a strictly defined model.
- This can be used to implement a user-interactive mechanism for hot-deployment (and removal) of aspects. For example, a long-running application could allow a user to apply a bugfix or extension patch at runtime via an aspect.
- XML-configuration mechanisms, as they are offered by some existing AOP tools (NAspect [Joh05], Spring [JH+05], JBoss [Bur04]) can be easily implemented by leveraging this mechanism.

Negative Adoptability Issues

- Runtime reconfiguration is somewhat restricted: only target instances created after the reconfiguration step are influenced, and one cannot add aspects to objects previously created.

  Comment It is true that the model is restricted by the definition that reconfiguration only affects subsequent object instantiations. However, this makes for a semantically sound model which circumvents many problems of self-modifying programs arising without such restrictions.

- Imperative runtime reconfiguration can lead to scattering and tangling of configuration statements, and it makes program analysis hard or even impossible.

  Comment Power comes at the price of complexity. Runtime reconfiguration should therefore only be used if absolutely necessary. Alternatively, runtime reconfiguration can be done in a temporary and localized way by undoing the changes after making the necessary object instantiations, as it was shown in listing 4.8.

4.3.3 Join Point Model

Building the base for the quantification part of an AOP tool (which is actually formed by the pointcut mechanism, see below), the join point model provided by an AOP implementation is one of its most important features. It defines the functionality available to an aspect for influencing the dynamic behavior of an application at runtime.

The join point model provided by XL-AOF is at the same time its biggest liability and an important asset. As was already discussed in chapter 3, an AOP tool built on top of a subclass proxy infrastructure can only implement a limited join point model, and while XL-AOF implements the full model provided by the infrastructure, it will always be inferior to any static and code weaving-based approach in this regard. On the other hand, XL-AOF provides a recursive join point model (aspects can be applied to other aspects), which is powerful and not very common, and it has a completely new feature of extensibility: it allows aspects to define (and trigger) new kinds of join points.

Join Point Kinds

Join points are very dynamic: every execution of a method at runtime, for example, can constitute a separate join point. However, it is possible to classify collections of join points based on their semantics of occurrence, and to give these *join point kinds* unique names. Of course, join points of different kinds occur at different places during program execution, at which different context information is available to an advice method. Similarly, some join points can have return values (e.g. a method execution), whereas in other situations, a join point will not return a value.

Therefore, we define different *signatures* for each join point kind: a signature comprises a list of *parameter definitions*, which specifies the context information available at a join point, as well as a *return type definition*. For example, *after returning* join points would have a common signature as indicated in listing 4.9: it comprises the target object whose method is to be executed, a reflective *MethodInfo* object, a list of the method's arguments, the value returned from the method, and a reflective *IJoinPointInfo* object comprising any additional, but less often used context information (for example, the number of advice methods already executed in the context of this join point or the current advice call stack; the full definition of this interface is given in a later section on join point introspection). The signature specifies a return value of type *object*, which indicates that the join point returns an (*object*) value.

```
object (object target, MethodInfo method, object[] args, object returnValue,
  IJoinpointInfo context)
```

Listing 4.9: *After returning* join point signature definition

In principle, a join point kind signature is built the same way as a method signature, and indeed the signature of a join point can be seen as the method signature of advice methods bindable to it (although XL-AOF defines some relaxed rules, which will be explained in section 4.3.4). Due to this property, and also because pointcuts need to identify the join point kinds they reference, it is necessary to have a way of expressing a join point kind definition in source code.

The type system of the .NET platform already includes a mechanism for defining signature-oriented types: *delegates*. Actually meant as a kind of type-safe function or method pointer (which can be used implement complex behaviors, such as instantiations of the *Command* or *Observer* patterns [GHJV95] in an easy way), they constitute a convenient way of integrating join point kind definition into a .NET-based AOP platform. XL-AOF therefore uses a unique delegate type for each join point kind it defines. For example, the *after returning* join point kind is defined as an *AfterReturning* delegate type, as shown in listing 4.10. The delegate's signature directly corresponds to the join point signature definition of listing 4.9.

```
public delegate object AfterReturning(object target, MethodInfo method, object[]
  args, object returnValue, IJoinpointInfo context);
```

Listing 4.10: *After returning* join point kind defined as a delegate

Listing 4.11 shows the whole set of join point kinds predefined by XL-AOF:

- *ObjectCreation* defines a kind of join point which encapsulates the creation of a target object instance—an aspect can advise this join point in order to influence how (and if) an object is actually created when a client uses *ObjectFactory.Create<>*. This can be used, for example, to implement object pooling aspects. The *creator* parameter should be used if the advice chooses to instantiate an object—it ensures that the instance is aspectized as needed.

- *BeforeConstruction* join points occur after *ObjectCreation* join points, just before a specific constructor is to be run. They can be used for parameter or security checks previous to executing a constructor, for example, but they do not have access to the constructed object yet.

- *AfterConstruction* join points occur immediately after an object instance has been successfully created—after the constructor has run, but before the instantiating code regains control. They can be used for post-initialization work.

- *BeforeMethod*, *AfterMethod*, and *AroundMethod* join points have the usual AspectJ semantics, as described in section 2.3.1 in chapter 2: they occur before, after, and around method executions (with around occurring between before and after join points).

  Note, however, that XL-AOF incorporates the *before*, *after*, and *around* properties into dedicated join point kinds rather than classifying advice with these properties (as AspectJ does). The reason for this is that we believe the concept is most meaningful only in conjunction with method execution join points (especially when considering *after returning* and *after exception*, see below) and should therefore be directly associated with these join point kinds rather than all advice methods.

  The *nextProceeder* parameter of the *AroundMethod* join point can be used by an advice to invoke the original method encapsulated by the join point (or the next advice, if several advice methods are bound to it).

- While *AfterMethod* join points always occur subsequent to a method execution (no matter whether successful or not), *AfterReturning* join points only occur (following the *AfterMethod* join points) if the respective method successfully returns a value. An advice bound to such a join point can inspect the return value and either return it directly to the caller, modify it before returning, or return a completely different value.

- *AfterException* join points occur instead of *AfterConstruction* or *AfterReturning* join points when a constructor or method throws an exception. Advice bound to such join points can handle the exception simply by returning a value (for constructors, the value should be *null*), in which case the usual *AfterReturning* and *AfterConstruction* join points are triggered (if these handlers throw an exception on their own, these do not trigger an *AfterException* join point for the executed method). An advice can also rethrow the exception passed to the join point as the *exception* property. This join point kind is useful for implementing error handling code associated with a cross-cutting concern.

```
public delegate object ObjectCreation(Type type, object[] constructorArgs,
  IObjectInstanceCreator creator, IJoinpointInfo context);
public delegate void BeforeConstruction(Type constructedType, ConstructorInfo
  constructor, object[] constructorArgs, IJoinpointInfo context);
public delegate void AfterConstruction(Type constructedType, object
  constructedObject, ConstructorInfo constructor, object[] constructorArgs,
  IJoinpointInfo context);

public delegate void BeforeMethod(object target, MethodInfo method, object[] args,
   IJoinpointInfo context);
public delegate void AfterMethod(object target, MethodInfo method, object[] args,
  IJoinpointInfo context);
public delegate object AroundMethod(object target, MethodInfo method, object[]
  args, IMethodProceeder nextProceeder, IJoinpointInfo context);

public delegate object AfterReturning(object target, MethodInfo method, object[]
  args, object returnValue, IJoinpointInfo context);
public delegate object AfterException(object target, MethodBase
  methodOrConstructor, object[] args, Exception exception, IJoinpointInfo
  context);

public interface IObjectInstanceCreator {
  object CreateInstance(Type type, object[] constructorArgs);
}

public interface IMethodProceeder {
  object Proceed(object[] args);
}
```

Listing 4.11: Join point kinds predefined by XL-AOF

There are no dedicated join point kinds for property get and set operations. This is because these operations are defined by the CLI specification as method executions [ECM05b], so they also trigger method execution join points. As will be explained in section 4.3.6, however, XL-AOF provides special support for accessing the property associated with a getter or setter method from within an advice bound to a method join point, and the pointcut mechanism allows an aspect to explicitly bind advice to getter or setter methods of specific properties.

Custom Join Points

One important feature of XL-AOF is that the set of join points available to an aspect is not fixed, but extensible: a programmer of an aspect-oriented application can easily define a new join point kind and trigger a join point of this kind at any time during object or aspect execution.

For example, a developer of an aspect which stores and loads object state to/from some persistence mechanism could use this feature to provide a docking point for other aspects: triggering *before data saved* and *after data loaded* custom join points just before the state is saved and just after it has been loaded would allow other aspects to extend this mechanism, for example by adding encryption or compression to the stored state data.

While the same would of course also be possible using object-oriented means (e.g. using inheritance or an *Observer* [GHJV95] implementation), the mechanism of declaring custom join points is both decoupled and *quantifiable* (see section 2.3.2). Decoupled means that neither aspect needs to know of the other one (in contrast, using inheritance makes the extending class depend heavily on the base class, using an *Observer* requires a common controller connecting the

extending and the base object), their only connection point is the custom join point. Quantifiable means that the extending aspect can use the quantification facilities (i.e. the pointcut mechanism) to bind to its targets, i.e. it can as easily be applied to a subset of the custom join point occurrences as to all of them.

Custom join points can also be triggered by aspect-aware base (i.e. non-aspect) code. Aspect-aware in this context means that the base code needs to be aware that it might be extended by aspects, it need of course not be aware of the concrete aspects applied to it.

Implementation From an implementation perspective, adding and triggering a custom join point requires the following steps:

- Define a delegate identifying the join point kind and its signature,
- Define and register a *join point handler* type, as explained below,
- Trigger the custom join points at the appropriate places during program execution.

Listing 4.12 shows an example of creating a new custom join point kind by declaring a new delegate type. It is identified by the delegate's name (*BeforeDataIsUsed*), and it is defined to take a string (*data*) as context information and to return a string to the triggering code.

```
public delegate string BeforeDataIsUsed(string data);
```

Listing 4.12: Custom join point kind definition

As the next step, a custom join point handler needs to be defined. A join point handler does the advice management for a specific join point kind: is keeps a list of advice methods bound to a join point kind, it determines which advice methods should be executed when a join point of that kind is triggered, and it executes the actual advice method, passing it the required context information and storing the return value.

Implementing a custom join point handler requires deriving a new class from the *JoinpointHandler<>* base class, and usually only involves overriding a single method: *ExecuteAdvice*, as illustrated in listing 4.13. *ExecuteAdvice* is called for each single advice which is executed at a specific join point occurrence. It is passed an object encapsulating the advice method (*advice*) and an object holding all the context information available at the time of triggering (*JoinpointInfo*), and it returns a value signaling whether the advice was actually executed. The *JoinpointInfo* object can be changed by the *ExecuteAdvice* method, for example if the advice changes the join point's return value, and advice methods executed subsequently for the same join point will receive the changed information.

In the listing, the advice method is executed with context information extracted from the *JoinpointInfo's AdditionalContext* property set (detailed in section 4.3.6), and its return value is stored in *joinpointInfo.ReturnValue*. In order to allow subsequent advice methods to react on the data returned by previously executed advice method, the context information is also updated.

After definition, the join point handler also needs to be registered, which is done declaratively with a *CustomJoinpoint* attribute. The attribute is applied to the delegate defining the custom join point kind and associates the custom join point type with the join point handler type. At runtime, the handler will automatically be instantiated for every aspect used in the assembly containing the attribute. Listing 4.13 thus extends listing 4.12 to contain all that is necessary to define a new custom join point kind.

```
class BeforeDataIsUsedHandler : JoinpointHandler<BeforeDataIsUsed> {
  protected override bool ExecuteAdvice(Advice<BeforeDataIsUsed> advice,
    JoinpointInfo joinpointInfo) {
    string contextInfo = (string)joinpointInfo.AdditionalContext["data"];
    joinpointInfo.ReturnValue = advice.AdviceMethod(contextInfo);
    joinpointInfo.AdditionalContext["data"] = joinpointInfo.ReturnValue;
    return true;
  }
}

[CustomJoinpoint(typeof(BeforeDataIsUsedHandler))]
public delegate string BeforeDataIsUsed(string data);
```

Listing 4.13: Custom join point handler definition

The last step, illustrated in listing 4.14, involves the triggering of the join point, which is performed via the *TriggerJoinpoint<>* method of the *AspectEnvironment* class. In the listing, the join point is triggered from a *LogString* method, which receives some data and logs it to the console. The method allows aspects to examine and possibly change the data prior to the logging, so it creates a *JoinpointInfo* object holding the context information (including join point type and *this* reference), triggers the join point, and uses the value returned by the aspects. After the triggering, the *JoinpointInfo* structure contains the return value of the last advice method being executed, but if no advice were executed, it wouldn't contain a value. Therefore, the method checks whether an advice was executed before using the return value.

```
void LogString(string data) {
  JoinpointInfo jp = new JoinpointInfo(typeof(BeforeDataIsUsed), this);
  jp.AdditionalContext["data"] = data;
  AspectEnvironment.TriggerJoinpoint<BeforeDataIsUsed>(jp);
  if (jp.NumberOfExecutedAdvice > 0) {
    data = (string)jp.ReturnValue;
  }
  Console.WriteLine(data);
}
```

Listing 4.14: Custom join point triggering

Discussion of Novelty To our knowledge, XL-AOF is the first AOP tool which allows new join point kinds to be added and join points to be explicitly triggered by aspect or base code. In part, this can compensate for the restricted join point model provided by the subclass proxy infrastructure; however it is not suitable for all join point scenarios (for example, in general it wouldn't be used to implement a *before field access* join point, since this would require custom join point triggering code before every single field access). On the other hand, the mechanism provides possibilities of semantic and domain-specific join points not offered by other AOP tools.

Positive Adoptability Issues

- XL-AOF's join point model offers the most important join points used in aspect-oriented programming: method execution, object-creation, and after object construction.
- XL-AOF offers the full join point model realizable with the subclass proxy infrastructure.
- With the possibility of providing custom join points, XL-AOF provides an extensible join point model not yet seen in any other AOP tool.
- Custom join points allow the extension of classic, predefined join point kinds by new, semantic (i.e. object- or aspect-defined) kinds, which allows more decoupled and flexible designs.

Negative Adoptability Issues

- The join point model provided by XL-AOF is restricted. For example, there are no field access join points, and only virtual method executions can be advised by an aspect.

  Comment Unfortunately, the subclass proxy infrastructure only allows a limited number of join point kinds to be triggered by the AOP tool. XL-AOF however does its best to circumvent these limitations: many situations can be remedied by using custom join points, others can be resolved by using method join points in conjunction with the right quantification. For example, instead of binding an advice to a field join point, a pointcut can be written to advise all *methods* which access the field, often with the desired results.

- Custom join points triggered from within base code require the code to be aware of aspects.

  Comment Aspect-awareness only means that the base code needs to know that it might be influenced by aspects, it doesn't couple the base code to any concrete aspects. However, triggering custom join points from base code means a tradeoff between obliviousness and expressiveness, and thus should only be used when the feature is really required.

4.3.4 Pointcuts and Advice

As the *quantification* part of AOP (see section 2.3.2 in chapter 2), the pointcut mechanism is one of the most important features of an aspect-oriented tool implementation: it allows the programmer to quantify over the set of available join points in an application, specifying the connection points between the cross-cutting concerns and the base code. Advice methods contain the aspect code which is executed when a join point picked by a pointcut is reached.

Pointcuts and advice are defined in a combined way in XL-AOF: advice methods are simply instance methods of the aspect class, and pointcuts are declaratively

applied to them using the *AdviceAttribute* in connection with a number of *filters*. The simplest form of an advice method is annotated only with the *AdviceAttribute* without filtering, as illustrated in listing 4.15. The *AdviceAttribute* takes the kind of join points to which the advice should be bound as a declarative parameter, so in the example, the advice will be bound to every *after returning* join point occuring on the aspect's target objects (selected via the aspect configuration, as described in section 4.3.2).

```
[Advice(typeof(AfterReturning))]
public object AnAdvice(object target, MethodInfo method, object[] args, object
  returnValue, IJoinpointInfo context) {
  Console.WriteLine("Execution of " + method.Name + " finished successfully.");
  return returnValue;
}
```

Listing 4.15: Simple advice definition

As already indicated in section 4.3.3, when an advice method is executed, it can access context information available for the current join point and it can return a value to the code triggering the join point. In the example listing, the advice method takes five parameters as context information from the *after returning* join point: the target object which executed the method, a reflective *MethodInfo* object, the method's arguments, the value it returned, and the *IJoinpointInfo* object holding any additional context information. The advice also has a return value, which is substituted for the join point's (i.e. method call's) own return value.

The advice can only have these parameters and return type because the join point declaration defines them. In other words, the delegate defining a join point kind also specifies the signature of the advice methods that can be bound to a join point of that kind. In detail, however, XL-AOF is very flexible what regards compatibility of an advice's signature with the join points it is bound to: as long as a mapping between advice arguments and join point context information can be unambiguously determined, and as long as the return types are compatible, the framework does not require the advice's signature to be exactly the same as the join point kind's signature.

In particular, an advice's parameters need not be in the order defined by the delegate's signature, it can omit a context information argument (unless it is an *out* argument), and advice parameters can have names and types different from those of the arguments in the delegate definition.

In order to determine the mapping between advice parameters and the context information provided by a join point, XL-AOF will first try to associate each parameter with a context information object by name. If none can be found that way, it looks for a context information object with a matching type. The advice method is compatible with the join point kind if (and only if) this lookup sequence unambiguously returns a matching context information argument (i.e. an argument with a compatible type) for each advice parameter, if there is a matching advice parameter for each *out* argument in the join point kind's specification, and if the advice's return type is compatible with the join point kind's return type.

Type compatibility between advice argument and context information is thereby defined using *contravariance*, i.e. an advice's parameter type must be the same

or more general than the type of the context information. Conversely, for the return value, *covariance* is used: the advice's return type must be the same or more specific than the return type of the join point. The types of *out* and *ref* parameters must be *invariant*, i.e. the advice parameter types are required to be exactly the same as the join point kind's parameter types.

Type compatibility for *in* parameters can also be forced by including an *AutoCastAttribute* declaration on the respective advice parameter declaration: if it is clear to the programmer that a certain context information parameter will always be of a more specific type than defined by the join point kind, the *AutoCastAttribute* will automatically cast (and box/unbox, as needed) the parameter at runtime, and contravariance is no longer enforced. This can, however, lead to runtime cast exceptions if the assumption was wrong.

To illustrate the possibilities offered by this definition of signature compatibility, consider listing 4.16, which shows three variations of the advice method defined in listing 4.15.

The first advice *(AdviceWithSignatureVariation1)* is equivalent to the advice of listing 4.15, but changes both the order and the names of some advice parameters. This is possible because XL-AOF can still find an unambiguous mapping to the join point kind's context information arguments by using the parameter types.

The second advice *(AdviceWithSignatureVariation2)* omits those parameters it doesn't make use of, which is possible because the respective context information arguments are not declared as *out* parameters.

The third advice *(AdviceWithSignatureVariation3)* assumes the return value of the advised join points will always be an integer and therefore includes an automatic cast declaration.

```
[Advice(typeof(AfterReturning))]
public object AdviceWithSignatureVariation1(IJoinpointInfo context, MethodBase
  myMethod, object target, object returnValue, object[] args) {
   Console.WriteLine("Execution of " + myMethod.Name + " finished successfully.");
   return returnValue;
}

[Advice(typeof(AfterReturning))]
public object AdviceWithSignatureVariation2(MethodBase myMethod, object
  returnValue) {
   Console.WriteLine("Execution of " + myMethod.Name + " finished successfully.");
   return returnValue;
}

[Advice(typeof(AfterReturning))]
public object AdviceWithSignatureVariation3(MethodBase myMethod, object target,
  [AutoCast]int returnValue, object[] args, IJoinpointInfo context) {
   Console.WriteLine("Execution of " + myMethod.Name + " finished successfully.");
   return returnValue;
}
```

Listing 4.16: Advice definition possibilities

Multiple Join Point Kinds Advice methods can be bound to multiple join point kinds at the same time by applying multiple *AdviceAttributes* to them. The semantics of such a combination is that of a logical *or*, i.e. the advice is bound to join points of either kind. Of course, the advice's signature must match the signatures of all join point kinds it is bound to.

Typesafe Advice

While XL-AOF's model of advice signature compatibility offers great flexibility, it has one important disadvantage: the rules of compatibility can only be enforced at runtime (when the aspect is used for the first time); no static type checking of the advice's signature is performed at compile-time. Usually, this dynamic checking will not be a problem, because unit-tests of the aspect will immediately show the problem. However, in some situations more static type safety will be desired. For these cases, XL-AOF provides an alternative way to combine pointcut and advice declaration in the aspect definition, as illustrated in listing 4.17.

```
[Pointcut]
public AfterReturning APointcut() {
  return delegate(object target, MethodInfo method, object[] args, object
    returnValue) {
    Console.WriteLine("Execution of " + method.Name + " finished successfully.");
    return returnValue;
  };
}
```

Listing 4.17: Typesafe advice declaration

As illustrated in the listing, a dedicated method declaration, attributed with a *PointcutAttribute*, is used to specify the pointcut. Its return type specifies the kind of join points selected by the pointcut, and its return value is a delegate holding the advice method. In this example, an anonymous delegate (i.e. an inline declaration of the advice method) is used for brevity, but the advice method could also be separately defined.

Compared with the equivalent pointcut/advice specification in listing 4.15, this approach is a little longer, it can't combine multiple join point kinds for one advice method, and it doesn't provide the signature-related flexibility offered by the dynamic way, but it has the great advantage of static type safety: the compiler will check signature compatibility at compile-time.

Discussion of Novelty To our knowledge, XL-AOF is currently the only AOP framework relying only on the standard compile-time tools (in fact, not including any demands for application compilation) providing a compiler-checked, type-safe advice declaration mechanism.

Filters

For both ways of pointcut/advice definition, filters are an important feature for selecting the join points targeted by an advice method. If a pointcut is simply bound to a join point kind such as *AfterReturning*, the respective advice methods will be invoked on every *after returning* join point occurring in the context of the aspect's target objects. To constrain this set to a number of specific join points, filters are used.

A filter is simply a custom attribute applied to the advice method definition (if the flexible *AdviceAttribute* approach is taken) or the pointcut method definition

(if the type-safe *PointcutAttribute* approach is taken). The attribute must implement the predefined *IJoinpointFilter* interface, whose straight-forward definition is given in listing 4.18: it contains just one method, which decides whether the pointcut matches a join point (given its *JoinpointInfo* context information object) or not.

```
public interface IJoinpointFilter {
  bool Matches(JoinpointInfo joinpoint);
}
```

Listing 4.18: *IJoinpointFilter* interface

XL-AOF predefines a number of useful filters, which comprise filtering on:

- Name of the join point *(WhereNameEqualsAttribute* and *WhereNameDiffersAttribute* for exact comparison, *WhereNameMatchesAttribute* and *WhereNameDoesntMatchAttribute* for regular expression matches),
- Declaring type of the join point *(WhereDeclaringTypeEqualsAttribute* and *WhereDeclaringTypeDiffersAttribute)*,
- Attributes defined on the join point *(WhereDefinedAttribute*, *WhereNotDefinedAttribute)*,
- Whether the method is a property's getter or setter method *(WhereGetterAttribute*, *WhereNotGetterAttribute*, *WhereSetterAttribute*, *WhereNotSetterAttribute)*, and
- What exception is thrown from a method *(WhereExceptionAttribute*; of the predefined join point kinds, this is only useful for *after exception* join points).

By default, the predefined filters check the actual join point for a match. For example, if the *WhereNameEqualsAttribute* is applied to a pointcut for *before method* join points, it checks whether the method's name equals a given string. In some situations, however, a different context would be desired. For example, as XL-AOF doesn't have explicit *property* join points but instead uses *method* join points, the context of a pointcut filter will often target the surrounding property rather than the method itself.

Therefore, some of the predefined filter attributes can be declaratively configured to match some specific context information of the join point. They have a *Context* parameter which takes values such as *Default*, *Method*, *Field*, *Type*, *Constructor*, or *Property*, which allow the programmer to select a specific context value if available. (If not, the filters will not match.)

Listing 4.19 shows an example pointcut/advice pair which binds to the *before method* join points of *getter methods* of properties, on which a specific *TagAttribute* is defined. It combines two filters with two different contexts: the *WhereGetterAttribute* is matched against the executed method, whereas the *WhereDefinedAttribute* is matched against the surrounding property.

```
[Advice(typeof(BeforeMethod))]
[WhereGetter]
[WhereDefined(typeof(TagAttribute), Context = Context.Property)]
void BeforePropertyGet(MethodInfo method) {
  Console.WriteLine("Property is retrieved.");
}
```

Listing 4.19: Advice targeting the getter method of specific properties

And-Combination of Filters If multiple filters are applied within a pointcut definition at the same time, the respective advice is only invoked if *all* of the filters match. This is equivalent to a logical *and* operation. The order in which the filters are evaluated is not specified, because the .NET specification does not define an ordering relation on custom attributes applied to a source code entity [ECM05b]. An *or*-combination of filters is not possible; it is necessary to define multiple separate pointcuts instead.

Control Flow Filters Sometimes, the advice method of an aspect should be executed only if not within the context of *another* advice method. For example, *reentrancy* of advice methods should often be avoided, a code example for which is given in listing 4.20. In the example, a *before method* advice is declared which prints the method name and the result of the target object's *ToString* method to the console. The problem with code such as this, which is a very common mistake made by AOP beginners, is that the advice is reentrant: the invocation of the target object's *ToString* method again invokes the advice, which leads to a (potentially) infinite recursion and (in reality) a *StackOverflowException*.

```
[Advice(typeof(BeforeMethod))]
public void Advice(object target, MethodInfo method) {
  Console.WriteLine("Entering method " + method.Name + " of " +
    target.ToString());
}
```

Listing 4.20: Advice definition with problematic reentrancy

While this could be remedied by constraining the advice to only trace methods whose name is different from *ToString*, the advice will then not log executions of the *ToString* method at all, and the protection is not very robust (i.e. the error reoccurs, if another method of the target object, e.g. its *Equals* method, is called).

A more robust pointcut would need to involve the current control flow (i.e. the execution stack) of the join point, only selecting those join points not located (directly or indirectly) within the *Advice* method. XL-AOF supports this scenario with predefined control flow filters, which allow the call stack to be checked before an advice is executed. XL-AOF provides the following filters:

- *WhereCFlowAttribute*, which executes an advice only if the control flow (i.e. the call stack) of the join point contains the given method or type, including the join point itself if it is a method call; its opposite *WhereNotCFlowAttribute*; as well as

- *WhereCFlowBelowAttribute* and its opposite *WhereNotCFlowBelowAttribute*, which are identical to *WhereCFlowAttribute* and *WhereNotCFlowAttribute*, but do not include the join point itself in the control flow check.

The difference between the *cflow* and the *cflow below* filters, which also exist in AspectJ with these semantics, is subtle, but important. If a filter is needed to include or exclude all join points regarding a certain method *and* all join points triggered (directly or indirectly) from within that method, the *cflow* filter is needed. If, on the other hand, the filter should ignore the join points for the method itself and handle only those join points triggered within the method, the *cflow below* filter is needed.

All control flow filters exist in variants taking a method name (for all methods with this name), a type and a method name (for specific methods only), and a type (for all methods of that type). Listing 4.21 shows the solution to listing 4.20's problem by checking that the control flow is not below a method named *Advice*.

```
[Advice(typeof(BeforeMethod))]
[WhereNotCFlowBelow("Advice")]
public void Advice(object target, MethodInfo method) {
  Console.WriteLine("Entering method " + method.Name + " of " +
    target.ToString());
}
```

Listing 4.21: Reentrancy problem solved via control flow filter

Custom Filters It is simple to implement custom filters simply by defining a custom attribute implementing the *IJoinpointFilter* interface. For the default join points, this will seldom be necessary, but for custom join points, this might well be reasonable.

As an example, consider again the *BeforeDataIsUsed* custom join point from listing 4.12 in section 4.3.3 and suppose this join point is to be used to implement a compression aspect, which compresses a string's data before it is used. A developer could choose that the compression algorithm is too ineffective for small amounts of data and should therefore be executed only for strings longer than 20 characters. Of course, this could be implemented manually by checking the string's length within the advice method, but this is not really where it belongs: it should be part of the quantification, i.e. of the aspect's pointcut definition.

Listing 4.22 therefore shows a custom filter definition implementing the described logic (while allowing the minimum length to be declaratively configured) and an application of the filter within a pointcut definition. The filter is implemented by accessing the context information stored within the *JoinpointInfo* object by the code triggering the join point (see listing 4.14 in section 4.3.3).

Ordering of Advice

If multiple advice methods are applied to the same join point, XL-AOF defines an order of execution only for *AfterException* join points: advice bound to such

```
public class WhereLengthAtLeastAttribute : Attribute, IJoinpointFilter {
  private int minLength;

  public WhereLengthAtLeastAttribute(int minLength) {
    this.minLength = minLength;
  }

  public bool Matches(JoinpointInfo joinpointInfo) {
    string data = joinpointInfo.AdditionalContext["data"] as string;
    if (data != null) {
      return data.Length >= minLength;
    }
    else {
      return false;
    }
  }
}

...

[Advice(typeof(BeforeDataIsUsed))]
[WhereLengthAtLeast(20)]
public string BeforeDataIsUsed(string data) {
  return packer.Compress(data);
}
```

Listing 4.22: Custom filter definition for a custom join point

a join point are executed as if they were catch handlers (i.e. handlers for derived exception types are executed before handlers of base exception types). For all other join points, and also for multiple *AfterException* advice for the same exception type, no ordering is defined.

This is not a problem in most scenarios, but sometimes, an aspect dependency requires a specific ordering. For this, XL-AOF allows programmers to prioritize aspects and advice methods in different ways by means of precedence and priority. Precedence is a relation between two aspects, where one aspect has precedence over another, meaning that the first aspect's advice methods are executed before (or around, in the case of *around method* advice) those of the second. Priority is a property of an advice method (which is relevant only after evaluation of aspect precedence): an advice method with a smaller numeric priority value is executed before an advice method with a larger priority value.

Precedence and advice can be defined as follows:

- An aspect developer can declare an aspect to have higher precedence than another aspect by applying a *PrecedenceAttribute* to the aspect's class definition which denotes the aspect with lower precedence,
- An aspect developer can include a *PriorityAttribute* in a pointcut specification in order to set the priority of the respective advice method,
- Any developer can include an assembly-level *GlobalAspectPrecedenceAttribute* into an application, which allows application-specific ordering of aspects, and
- Any developer can express precedence of an aspect over another for a specific target object at the time of aspect configuration; this is achieved via applying an *AspectPrecedenceAttribute* for type-level configuration, by

setting a declarative parameter of the *GlobalAspectBindingAttribute* for assembly-level configuration, and by using a respective overload of the *AttachAspect<>* method for dynamic configuration (see section 4.3.2).

Specifying aspect precedence can easily lead to problems, e.g. circular dependencies, which cannot be resolved at runtime, and defining aspect dependencies adds an amount of complexity to an application. It is therefore important to use the precedence and priority features only if they are really necessary; ideally, independent cross-cutting concerns should not have any influence on each other.

Positive Adoptability Issues

- XL-AOF provides a simple way of defining advice methods simply as methods annotated with an *AdviceAttribute*.
- Due to the model of using delegate types to identify join point kinds, it is easy and typesafe to select a specific join point kind for a pointcut simply by passing the type as a parameter to the *AdviceAttribute*. Typing errors are immediately detected by the compiler, and IDE typing support ("intellisense") is available.
- The default way of defining advice methods provides a great deal of signature flexibility: an advice can choose which context information it needs and which it doesn't, and it can rename a join point's contextual arguments if necessary.
- It is possible to use an advice method for different join point kinds (or-combination).
- For situations where unit tests are not available for aspects, XL-AOF provides a different, compiler-checked (yet less flexible) way of pointcut/advice declaration.
- Declarative, and-combinable filters offer a readable and powerful way of narrowing the set of joinpoints selected by a pointcut.
- A number of filters is predefined for common pointcut expressions, and it is easy to define new ones.
- XL-AOF provides great flexibility for aspect ordering on precedence and priority basis. Precedence can be defined at aspect development time as well as at configuration time.

Negative Adoptability Issues

- The flexible way of advice definition isn't compiler-checked, whereas the compiler-checked way is not flexible.

  Comment Due to the fact that XL-AOF needs to rely on existing compilers, it is not possible to implement a flexible *and* compiler-checked model for advice definition. In fact, we are quite proud that we have found a way to exploit anonymous delegates to include a compiler-checked model at all, even though it is less flexible than the default one.

- Filters are only evaluated at runtime; i.e. it is not possible to evaluate whether a filter is faulty (e.g. matches no target methods) at compile-time.

  Comment This is also by design, due to the fact that XL-AOF comes without a compiler. We've tried, however, to work with compiler-checkable entities as often as possible. For example, we use *Type* objects and enumerations instead of relying on (error-prone) string literals wherever possible, as opposed to many other aspect-oriented tools.

- Aspect precedence can lead to errors and inconsistencies.

  Comment This is true, and precedence and priority should therefore only be used if absolutely necessary.

4.3.5 Interface Introduction

As explained previously, aspect-oriented tools should not only be able to influence the dynamic behavior of an application, but its static structure as well. While the join point/pointcut/advice mechanism targets the former requirement, XL-AOF also provides an *introduction* (or *intertype declaration*) mechanism for the latter.

In section 3.1.2 in chapter 3, we explained that the only kind of introduction implementable by runtime approaches with statically compiled languages is interface introduction, which is enabled through dynamic type casts supported by such languages. XL-AOF provides interface introduction by means of the declarative *IntroduceAttribute*.

Usage of introduction is very simple: if an aspect should add an interface implementation to its target class, this interface implementation must be provided in the form of a nested type or a field (*"introducer"*) attributed with the *IntroduceAttribute*.

Listing 4.23 shows two equivalent aspects adding the .NET-defined interface *ICloneable* to a target object, the first implemented via a nested class, the second with a field as an introducer. While the first version is somewhat more compact, the second version is useful in situations where existing classes should be used as introducers. In addition, it allows other code to access the introducer instance, which can be useful in some situations. In both versions, cloning is implemented by calling the protected *MemberwiseClone* method implemented by the *System.Object* base class.

The aspects use introspection and field injection in order to access the target object and to call the protected method, this will be further discussed in section 4.3.6. For this section, it will suffice to say that introducers can use exactly the same features of aspect-related introspection and injection as the surrounding aspect can. In addition, XL-AOF optionally allows introducers defined as nested types of an aspect to take a reference to the aspect as a constructor parameter—XL-AOF will automatically pass a reference to the surrounding aspect via that parameter when it instantiates the introducer.

```
public class CloneableAspect1 {
  [Introduce]
  public class CloneableIntroducer : ICloneable {
    [Target] private object target = ReflectedField<object>.InjectedTarget;

    public object Clone() {
      return AspectEnvironment.Call<object>(target, "MemberwiseClone");
    }
  }
}

public class CloneableAspect2 {
  [Introduce]
  private CloneableIntroducer introducer = new CloneableIntroducer();
}

public class CloneableIntroducer : ICloneable {
  [Target]
  private object target = ReflectedField<object>.Injected;

  public object Clone() {
    return AspectEnvironment.Call<object>(target, "MemberwiseClone");
  }
}
```

Listing 4.23: Implementing *ICloneable* via introduction

Positive Adoptability Issues

- XL-AOF provides a very simple and easily understandable system of interface introduction via *introducers*: nested types or fields implementing the introduced interface and annotated with the *IntroduceAttribute*.
- Interface introduction is very flexible: introducers can employ the same means of introspection and field injection as the surrounding aspects can.
- By using the underlying mixin technology provided by *DynamicProxy*, interface introduction is very fast.

Negative Adoptability Issues

- Introduction can only add interfaces, not fields or single methods. XL-AOF also doesn't provide a mechanism for changing base classes, as some other AOP tools do.

  Comment As a runtime approach for statically type-checked programming languages, interface introduction is the only kind of introduction implementable, as discussed in chapter 3, section 3.1.2.

4.3.6 Aspect-Related Introspection

Runtime introspection of objects, also called *reflection*, is an important feature on modern object-oriented platforms, described for example by Jeffrey Richter in his book "CLR via $C^{\#}$" [Ric06]. .NET provides introspection support with the *System.Reflection* namespace, and XL-AOF employs this namespace for a large part of its functionality; e.g. for analyzing the attributes defined on classes

and methods, and for retrieving the *MethodInfo* context information objects of method-related join points. The underlying subclass proxy infrastructure uses the reflection extension *Reflection.Emit* to generate the subclass proxies XL-AOF's weaving is based on.

Due to the dynamic code generation transparently performed by XL-AOF at runtime, there are two important caveats when using the .NET-provided reflection mechanisms in an AOP-enabled scenario:

- Type equality checks must be avoided: direct type tests as illustrated in listing 4.24 will not work in AOP scenarios. Instead, polymorphism-enabled *assignability checks* should be used, as shown in listing 4.25.
- Member lookups might not work with an object created via the *ObjectFactory*, although they do when the object is created via *new*. Like with type tests, lookups such as *<object>.GetType().GetMethod(<methodname>)* should be avoided, inheritance-enabled lookups should be used instead.

```
if (<object>.GetType() == typeof(<Class>)) {
  ...
}
```

Listing 4.24: Direct type test failing in AOP scenarios

```
if (<object> is <Class>)} {
  ...
}

if (typeof(<Class>).IsAssignableFrom(<object>.GetType())) {
  ...
}
```

Listing 4.25: Polymorphism-enabled type test working well in AOP scenarios

Both problems are based on the fact that the dynamic type of an object needn't be the type passed to the *ObjectFactory.Create<>* method, and also arise in ordinary object-oriented programming scenarios as soon as polymorphism is employed. Since both issues have simple work-arounds, they aren't limiting, but should be known by programmers wishing to make use of instrospection in AOP-enabled scenarios.

In parallel to the .NET-provided reflection mechanism, XL-AOF also provides additional mechanisms for aspect-related introspection, which will be discussed in the following.

Retrieving Factory Attributes

In some cases, it can be important to be able to retrieve all the factory attributes involved in the aspect configuration of a class. For example, if a certain aspect binding should be removed from a type via XL-AOF's dynamic aspect configuration mechanisms (see section 4.3.2), it is necessary to first retrieve the exact factory attribute instance constituting that binding.

For this, the *ObjectFactory* class provides a method called *GetFactoryAttributes<TTarget>*, which returns all factory attribute instances for a given target

type. These instances include those directly declared on the type, those included via assembly-level configurations, and those manually added via calls to *AttachAttribute*.

Join Point Introspection with *IJoinpointInfo*

As shown in section 4.3.3, a *JoinpointInfo* object (in conjunction with a join point handler) is used to pass information from the context of a join point to an advice method. All join point kinds preimplemented by XL-AOF are defined in such a way that advice methods bound to such join points can request to receive a read-only version (*IJoinpointInfo*) of that object. While most context information is directly available in the form of advice parameters, information of less importance or information which is expensive to create can only be accessed via the *IJoinpointInfo* object.

In particular, the *IJoinpointInfo* object provides access to the kind of the join point which triggered the advice, the name of the join point (if available), the custom attributes associated with the join point, the number of previously executed advice methods, and the call stack at the time of advice execution (which is also used by the control flow filters, see section 4.3.4). In addition, custom join points can add customized introspective information by using an *IJoinpointInfo.AdditionalContext* name/object repository.

The full definition of the *IJoinpointInfo* interface is given in listing 4.26.

```
public interface IJoinpointInfo {
  Type Kind { get; }
  string Name { get; }
  object[] Attributes { get; }
  int NumberOfExecutedAdvice { get; }
  Stack<MethodInfo> CallStack { get; }
  IDictionary<string, object> AdditionalContext { get; }
}
```

Listing 4.26: *IJoinpointInfo* interface

AspectEnvironment Introspection

For common aspect-related instrospection requirements, the *AspectEnvironment* class offers the following methods:

- *public static T AspectOf<T>(object o):* Returns an aspect instance of type *T* applied to the object *o*, or *null* if no such aspect instance exists.
- *public static Type GetOriginalBaseType(object o):* Returns the original, unaspectized type of the object *o*, i.e. the type passed to *ObjectFactors.Create<>* when the object *o* was created.
- *public static Type GetOriginalBaseType(Type t):* Like *GetOriginalBaseType(object o)*, this returns the original, unaspectized type of an object, but this method is given the object's dynamic type rather than the object itself.

- *public static ReflectedProperty<TValue> PropertyOf<TValue>(object target, MethodInfo getterOrSetter):* This method returns the property (of type *TValue*) from the getter or setter method of an object *target*. While not specifically related to AOP, this is very useful when writing get/set advice methods for properties. An example usage for this is given in section 4.4.6.

- *public static ParameterCollection GetParameterCollection(MethodInfo method, object[] args):* The predefined method-related join points provide both a reflective *MethodInfo* object and an array of parameter values. Often, however, more sophisticated methods of reflective access to parameters are needed. For such situations, this method creates a *ParameterCollection* for the given method and argument values, which holds a set of reflective *Parameter* objects, allowing enumeration over the parameters, or direct access by index and by parameter name. The *Parameter* objects allow simple access to a parameter's name, its value, its custom, attributes, and its type. While the utility of this class is not restricted to aspect-oriented applications, it makes implementation of method-related advice much easier.

Injected Fields and Properties

In addition to manual introspection, XL-AOF also supports a mechanism for automatic introspective field or property injection for aspects. Injection means that the aspect declares a special reflective member, which is filled by XL-AOF at runtime. In particular, XL-AOF can inject references to the target object of an aspect as well as to fields and properties of the target object, as illustrated in figure 4.1.

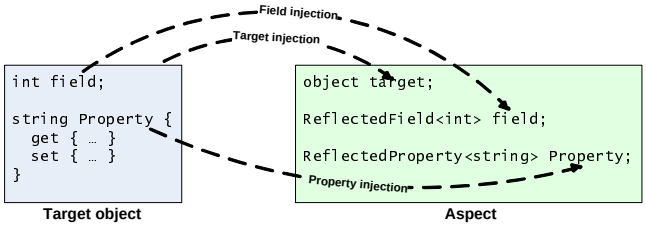

Figure 4.1: Injections supported by XL-AOF

Injections are performed just after the object has been created by the *ObjectFactory* and before any *after construction* advice is executed. Note that the semantics of injections are only meaningful for aspects having exactly one target object. For aspects shared by different target objects, the injected fields will always reference the target object created last and can therefore only be relied upon in *after construction* advice methods.

Target Injection As illustrated in listing 4.27, a *TargetAttribute* can be applied to fields of an aspect in order to instruct XL-AOF to inject a reference to the target object into that field. XL-AOF defines an *InjectedTarget* constant in

the *ReflectedField<>* class which should be assigned to injected fields in order to making it clear to the reader and the compiler that the field is injected by XL-AOF—this improves readability and suppresses any compiler warnings over the field not being assigned a value.

```
class Aspect {
  [Target] private object targetObject = ReflectedField<object>.InjectedTarget;

  [Advice(AfterConstruction)]
  public void Advice(object constructedObject) {
    Debug.Assert(constructedObject == targetObject);
  }
}
```

Listing 4.27: *TargetAttribute* for injecting a reference to the target object

Field and Property Injection Often, an aspect needs access to a target object's properties or, due to the cross-cutting nature of its concern, even to the object's private state. For these cases, XL-AOF allows an aspect to declare fields of type *ReflectedField<>* and *ReflectedProperty<>*, which are automatically injected with references to a field or property of the target object. The declared fields must either be called the same as their respective target fields or properties, or they must be attributed with a *SourceNameAttribute*. Listing 4.28 shows this for field injection: both reflected fields shown in the example equivalently reference the integer field "dataField" of the target object; property injection works equivalently. For documenting that the contents of the field will be injected at runtime, the constant *Injected* provided by XL-AOF should be assigned to the field in the source code.

As explained in section 3.2 in chapter 3, *ReflectedField<>* can be implemented very efficiently, while providing access to public as well as private and protected fields of the target object. Get and set operations conducted on an *ReflectedProperty<>* object simply result in calls to the property's getter and setter methods and are thus also very efficient.

```
class Aspect {
  ReflectedField<int> dataField = ReflectedField<int>.Injected;
  [SourceName("dataField")] ReflectedField<int> secondReflectedField =
    ReflectedField<int>.Injected;

  [Advice(typeof(BeforeMethod))]
  void BeforeAdvice() {
    Console.WriteLine(dataField.Value);
    ++secondReflectedField.Value;
    Debug.Assert(dataField.Value == secondReflectedField.Value);
  }
}
```

Listing 4.28: *ReflectedField<>* members for injecting references to fields

Field and Property Collection Injection If an aspect needs access to a whole row or all of the target object's fields or properties, *ReflectedField<>* and *ReflectedProperty<>* might be cumbersome, especially with a high number of target members. If the target members' names or number are unknown at compile time, it's even impossible to use single member injection. For example,

a serialization aspect would need to access all fields of its target object in a uniform way, no matter how many there are and how they are called.

For this, XL-AOF can inject references to all or a specific subset of the target object's fields or properties into aspect fields of types *ReflectedFieldCollection<>* and *ReflectedPropertyCollection<>*, as illustrated in listing 4.29 for fields (property injection works in an analoguous way). The collection objects allows the injected fields and properties to be enumerated or accessed by name. By default, all fields or properties are injected; to select a specific subset, the same filters used for pointcuts, such as *WhereDefinedAttribute* or *WhereNameMatchesAttribute*, can be used. In the listing, the *WhereDefinedAttribute* filter is used to only inject those fields which are tagged with a specific attribute.

Similar to the single member injections, *ReflectedFieldCollection<>* and *ReflectedPropertyCollection<>* provide an *Injected* constant to improve readability and avoid compiler warnings.

```
class Aspect {
  ReflectedFieldCollection<object> allFields =
    ReflectedFieldCollection<object>.Injected;

  [WhereDefined(typeof(Tag))]
  ReflectedFieldCollection<int> taggedFields =
    ReflectedFieldCollection<int>.Injected;

  [Advice(typeof(BeforeMethod))]
  void BeforeAdvice() {
    foreach (ReflectedField<object> field in allFields) {
      Console.WriteLine(field.Value.ToString());
    }
    ++taggedFields["dataField"];
  }
}
```

Listing 4.29: *ReflectedFieldCollection<>* for injecting of a whole set of fields

Positive Adoptability Issues

- XL-AOF does not restrict the use of .NET reflection in AOP-enabled applications.
- Aspect-related introspection is provided by an additional reflection mechanism provided by the framework.
- Join point context information can be inspected using the *IJoinpointInfo* context object.
- The *AspectEnvironment* class provides useful aspect-level reflection such as returning the aspects attached to a class, and it provides convenient support for writing method-based advice for properties.
- Target, field, and property injection can be used to provide an aspect with references to the target object, its properties, and even its internal state.

Negative Adoptability Issues

- There is a .NET reflection caveat for objects created with the *ObjectFactory* concerning the dynamic type of object.

  Comment The same caveat applies for ordinary code making use of object-oriented polymorphism, and should therefore be considered all the time.

- Access to private fields of objects can be considered a breach of data hiding.

  Comment With traditional object-oriented programming, the concerns implemented by aspects are part of the object itself and therefore have access to the object's private state. Encapsulating the concern as an aspect often cannot remove this coupling. Ideally, there should be an accessibility level comparable to *protected* explicitly allowing access to a field only to aspects applied to a class, but unfortunately the .NET specification does not provide such an accessibility level. Access to private fields is therefore necessary.

4.3.7 Aspect Dependencies

While aspects should in all cases be as decoupled as possible from each other, there are many situations where aspects have influence on each other, and some cases where they have to interact. Aspect interactions are a complex research topic, whose current state-of-the-art has been summarized by Samen et al in the AOSD-Europe deliverable *Study on interaction issues* in early 2006 [STW+06]. The topic of aspect interactions would go beyond the scope of this thesis, but XL-AOF supports the interaction kind denoted by Samen et al as *Dependency*—an aspect requiring another aspect to be applied to the same target.

With XL-AOF, this requirement can be expressed by applying an instance of *RequiresAspectAttribute* to the aspect's class. *RequiresAspectAttribute* takes an aspect type and a factory attribute type (as well as constructor arguments for it) as its parameters, and XL-AOF guarantees that the factory attribute is used to instantiate an aspect on every target class, unless an aspect of the given type is already applied to it. For convenience, specializations of *RequiresAspectAttribute* already exist for the predefined singleton, per object, and per class factory attributes.

For better illustration, consider listing 4.30. In this example, an aspect *Aspect1* is declared, which depends on an aspect *Aspect2*. Whenever *Aspect1* is applied to a target class, XL-AOF will analyze whether an instance of *Aspect2* has already been applied. If not, it will use the *SingletonAspectAttribute* factory attribute in order to instantiate *Aspect2*. For simplification, the listing uses the *RequiresSingletonAspectAttribute*, which is equivalent to an *RequiresAspectAttribute* configured to use the *SingletonAspectAttribute*.

```
[RequiresSingletonAspect(typeof(Aspect2))]
class Aspect1 {
  ...
}
```

Listing 4.30: Aspect dependency with standard attribute

Positive Adoptability Issues

- Dependencies are a very common form of aspect interaction. Because XL-AOF provides predefined support for them, many scenarios where one aspect uses or extends structure or behavior added by another aspect can be effortlessly implemented.

4.4 Tutorials and Samples

Concluding this section about XL-AOF, we will now give a number of introductory examples for cross-cutting concerns solved using XL-AOF, presented in a tutorial-like fashion. These concerns are of general purpose and not directly related to distributed or space-based computing yet, but they are aimed to make readers more acquainted with XL-AOF's features, and should lead to a better understanding of the more complex distribution-oriented aspects given in the following chapters.

4.4.1 Motivating Example: Accounts

As a motivating example for the tutorials, we choose a very simple scenario, which allows for integration of many different cross-cutting concerns: a banking application. In particular, we will concentrate on one single class of the application: *Account*, which is illustrated in listing 4.31. The class encapsulates information about one single bank account, which has an ID, an owner, and a balance as its state information. It allows for money to be deposited, withdrawn, and transfered, the latter of which consists of a joint deposit and withdraw operation.

```
public class Account {
  private string id;
  private decimal balance;
  private string owner;

  public Account(string id, decimal initialBalance, string owner) {
    this.id = id;
    this.balance = initialBalance;
    this.owner = owner;
  }

  public virtual string ID {
    get { return id; }
  }

  public virtual decimal Balance {
    get { return balance; }
    protected set { balance = value; }
  }

  public virtual string Owner {
    get { return owner; }
    set { owner = value; }
  }

  public virtual void Deposit(decimal amount) {
    this.Balance += amount;
  }
```

```
  public virtual void Withdraw(decimal amount) {
    this.Balance -= amount;
  }

  public virtual void Transfer(decimal amount, Account to) {
    to.Deposit(amount);
    this.Withdraw(amount);
  }

  public override string ToString() {
    return Owner + "'s account, current balance: USD " + balance.ToString("N");
  }
}
```

Listing 4.31: *Account* class used as a base for the following tutorials

To illustrate the use of this account, we will create a test driver simulating interactions with accounts conducted by two fictional characters called *Homer* and *Jack*. Consider the test driver shown in section 4.32, which first creates an account for Homer, who then deposits and withdraws certain amounts. Next, an account for Jack is created, and Homer transfers an amount of money to Jack's account.

```
Account homer = ObjectFactory.Create<Account>("10242083", 1000m, "Homer");
homer.Deposit(100.0m);
homer.Withdraw(100.0m);

Account jack = ObjectFactory.Create<Account>("213456757", 2000m, "Jack");
homer.Transfer(1532.95m, jack);
```

Listing 4.32: Simple test driver for the *Account* class

In the following, we will give a number of additional (and cross-cutting) requirements for the *Account* class and explain how these can be implemented using XL-AOF. After that, we will ask the question of the exact advantages of the aspect-oriented solutions we present as compared to a traditional solution and give a short summary of the advantages and disadvantages of the aspect-oriented implementations of the particular concern.

4.4.2 Concern 1: Method Tracing

When executing the test driver, it can be noted that nothing whatsoever is printed to the console. In order to ensure the correct workings of the *Account* class, some sort of diagnostic information should be displayed on the screen. In fact, it would be nice to just get a trace of all operations executed on account objects and of their states when running the test driver. Such method tracing—a classical cross-cutting concern—of course should be implemented as an aspect.

By using XL-AOF, we can write a perfectly reusable tracing aspect, which is completely oblivious of its concrete target objects; we'll call the respective class *TraceAspect*. For starters, *TraceAspect* should simply write a log message to the console whenever an operation is invoked on an account object. We therefore add an advice method bound to *before method* join points within the *TraceAspect* class, as shown in listing 4.33.

```
public class TraceAspect {
  [Advice(typeof(BeforeMethod))]
  public void LogActionStart() {
    Console.WriteLine("Operation starting!");
  }
}
```

Listing 4.33: First attempt at logging aspect definition

In order to try out the aspect, we need to configure the aspect to be applied to the *Account* class. We could do that by applying a factory attribute to *Account*. However, tracing is a concern which shouldn't be too tightly bound to the target class. For example, a requirement might be to switch on and off logging without changing *Account's* source code, or to have all debugging aspect configurations located in one common place. Therefore, we will supply an assembly-level configuration attribute; and because the *TraceAspect* does not store any instance data, we can use the predefined *GlobalSingletonAspectAttribute*. Listing 4.34 shows both aspect configuration and aspect definition, and listing 4.35 shows the console output generated by the test driver.

```
[assembly: GlobalSingletonAspect(typeof(Account), typeof(TraceAspect))]
...
public class TraceAspect {
  [Advice(typeof(BeforeMethod))]
  public void LogActionStart() {
    Console.WriteLine("Operation starting!");
  }
}
```

Listing 4.34: Logging aspect with configuration

```
Operation starting!
Operation starting!
Operation starting!
Operation starting!
Operation starting!
```

Listing 4.35: Output of the first logging aspect

The output is encouraging: the aspect seems to work correctly. However, it's not very useful yet, as it doesn't provide any usable information. We will therefore change the advice method to include a dump of the operation's method name and the target object's string representation, as illustrated in listing 4.36.

```
[assembly: GlobalSingletonAspect(typeof(Account), typeof(TraceAspect))]
...
public class TraceAspect {
  [Advice(typeof(BeforeMethod))]
  public void LogActionStart(object target, MethodInfo method) {
    Console.WriteLine("STARTING: " + method.Name + ", target: " +
      target.ToString());
  }
}
```

Listing 4.36: Second attempt at logging aspect definition

This, however, now yields a *StackOverflowException*; what went wrong? The explanation is simple: the *ToString* method, invoked from the *LogActionStart* advice, is also handled by the advice, which results in an endless loop and, eventually, a *StackOverflowException*.

To remedy this, we will simply include a filter in the advice's pointcut: by using a *control flow* restriction, we can ensure that the *ToString* method itself and any methods called from within it will not trigger the advice's execution. The working aspect is shown in listing 4.37, and its output is displayed in 4.38. It uses a *cflow* filter rather than a *cflow below* one; the effect of this—apart from stopping the infinite loop—is that the trace log doesn't show any calls to the *ToString* method.

```
[assembly: GlobalSingletonAspect(typeof(Account), typeof(TraceAspect))]
...
public class TraceAspect {
  [Advice(typeof(BeforeMethod))]
  [WhereNotCFlow("ToString")]
  public void LogActionStart(object target, MethodInfo method) {
    Console.WriteLine("STARTING: " + method.Name + ", target: " +
      target.ToString());
  }
}
```

Listing 4.37: Second, corrected attempt at logging aspect definition

```
STARTING: Deposit, target: Homer's account, current balance: USD 1.000,00
STARTING: Withdraw, target: Homer's account, current balance: USD 1.100,00
STARTING: Transfer, target: Homer's account, current balance: USD 1.000,00
STARTING: Deposit, target: Jack's account, current balance: USD 2.000,00
STARTING: Withdraw, target: Homer's account, current balance: USD 1.000,00
```

Listing 4.38: Output of the second logging aspect

After that, it's trivial to fully make *TraceAspect* a useful debugging tool: listing 4.39 adds *after returning* and *after exception* advice methods to the aspect and fine tunes it not to log any property accessors, and listing 4.40 shows the respective console output.

```
[assembly: GlobalSingletonAspect(typeof(Account), typeof(TraceAspect))]
...
public class TraceAspect {
  [Advice(typeof(BeforeMethod))]
  [WhereNotCFlow("ToString")]
  [WhereNotSetter, WhereNotGetter]
  public void LogActionStart(object target, MethodInfo method) {
    Console.WriteLine("STARTING: " + method.Name + ", target: " +
      target.ToString());
  }

  [Advice(typeof(AfterReturning))]
  [WhereNotCFlow("ToString")]
  [WhereNotSetter, WhereNotGetter]
  public object LogActionEnd(object target, MethodInfo method, object
    returnValue) {
    Console.WriteLine("FINISHED: " + method.Name + ", target: " +
      target.ToString());
    return returnValue;
  }

  [Advice(typeof(AfterException))]
  [WhereNotCFlow("ToString")]
  [WhereNotSetter, WhereNotGetter]
  public object LogActionException(object target, MethodBase method, Exception
    exception) {
    Console.WriteLine("FAILED: " + method.Name + ": " + exception.Message);
    throw exception;
  }
}
```

Listing 4.39: Third, complete attempt at logging aspect definition

```
STARTING: Deposit, target: Homer's account, current balance: USD 1.000,00
FINISHED: Deposit, target: Homer's account, current balance: USD 1.100,00
STARTING: Withdraw, target: Homer's account, current balance: USD 1.100,00
FINISHED: Withdraw, target: Homer's account, current balance: USD 1.000,00
STARTING: Transfer, target: Homer's account, current balance: USD 1.000,00
STARTING: Deposit, target: Jack's account, current balance: USD 2.000,00
FINISHED: Deposit, target: Jack's account, current balance: USD 3.532,95
STARTING: Withdraw, target: Homer's account, current balance: USD 1.000,00
FINISHED: Withdraw, target: Homer's account, current balance: USD -532,95
FINISHED: Transfer, target: Homer's account, current balance: USD -532,95
```

Listing 4.40: Output of the third logging aspect

Comparison

Implementing method tracing directly within the *Account* class would have required instrumentation of every method with calls to the *Console.WriteLine* method. If exceptions should be traced, which isn't needed at the moment but might be with the changes made by the subsequent tutorial sections, *try/catch* blocks would be required, as illustrated in listing 4.41 for the *Transfer* method.

```
public void Transfer(decimal amount, Account to) {
  Console.WriteLine("STARTING: Transfer, target: " + ToString());
  try {
    to.Deposit(amount);
    this.Withdraw(amount);
  }
  catch (Exception exception) {
    Console.WriteLine("FAILED: Transfer: " + exception.Message);
    throw;
  }
  Console.WriteLine("FINISHED: Transfer, target: " + ToString());
}
```

Listing 4.41: *Transfer* method tangled with tracing code

Advantages of the Aspect-Oriented Solution

Code locality and understandability: With the better separation of concerns achieved by encapsulating the tracing concern as an aspect, code locality is improved: code dealing with the logging of method starts is located in one dedicated (advice) method, as is method return and exception logging code. Each method of the *Account* class also deals with exactly one (functional) concern, such as depositing, withdrawing, and transferring money. This locality of code is beneficial for readability and understandability of the code—in order to understand how a concern is dealt with, just one single method (possibly advice) needs to be considered. In contrast, with the object-oriented solution, different concerns are spread across several methods, and every method contains several concerns, which makes the code much harder to understand.

Reusability of the target class: In the aspect-oriented implementation, the *Account* class can easily be reused in situations where no logging should be performed, without making any changes to its class definition. To reuse the object-oriented implementation in such a scenario means to manually untangle and remove all logging code from every single method of *Account*.

Reusability of the aspect: The tracing aspect is also highly reusable; nothing in its code depends on implementation details of the *Account* class. Therefore, it can be applied to other target classes without making any changes to the aspect's code. With the object-oriented version, new tracing instrumentation has to be inserted into every new target class by hand.

Scalability of the aspect: In the aspect-oriented version, tracing is automatically added to every new method of the *Account* class without any additional work and to every other class with only minimal effort. In contrast, considerable effort would have to be taken to make new methods/classes traceable with an object-oriented implementation.

Changeability of the class: The functionality of the *Account* class can be changed easily without affecting the tracing concern. In contrast, with a tangled object-oriented implementation, each change to the *Account's* functionality would need special care to be taken in order not to change the tracing concern's semantics.

Changeability of the aspect: In the aspect-oriented version, changes in the tracing behavior can be performed without touching the *Account* source code, which is not possible with an object-oriented implementation.

Configurability: Tracing can be separately implemented and declaratively configured in a single, central place, without touching either aspect or *Account* source code. Using XL-AOF's runtime configuration mechanism, tracing can even be added, removed, and reconfigured at runtime. All this is not possible with an object-oriented solution, which needs the tracing code to be written together with the *Account* class.

Disadvantages of the Aspect-Oriented Solution

***ObjectFactory* and virtual methods:** All objects to be traced must adhere to the design restrictions implied by XL-AOF: they must be created using the *ObjectFactory* and methods used as join points must be virtual. These restrictions would not apply to an object-oriented version.

CFlow filter necessary: In order to avoid a *StackOverflowException*, a somewhat complex control flow filter is necessary. In an object-oriented implementation, the stack overflow potential would be obvious (since *ToString* would be called from the logging code in *ToString*) and would thus be easily avoided.

4.4.3 Concern 2: Declarative Parameter Checking

An important weakness of the existing *Account* implementation is easily demonstrated by extending the test driver of listing 4.32: as the tracing aspect tells us, Homer's account is overdrawn by about 533 dollars after the test driver's code. Homer could therefore try to outbalance it by outwitting implementation as shown in listing 4.42—he first tries withdrawing a negative amount of money and then transfers 600 dollars from a fake account without owner. (The *TryOut*

method simply swallows any exceptions thrown by the code executed within its context.)

```
Account homer = ObjectFactory.Create<Account>("10242083", 1000m, "Homer");
homer.Deposit(100.0m);
homer.Withdraw(100.0m);

Account jack = ObjectFactory.Create<Account>("213456757", 2000m, "Jack");
homer.Transfer(1532.95m, jack);

TryOut(delegate {
  homer.Withdraw(-600m);
});

TryOut(delegate {
  Account falseAccount = ObjectFactory.Create<Account>("fake", 600m, null);
  falseAccount.Transfer(6000m, homer);
});
```

Listing 4.42: Outwitting the *Account* class

When running this code, it works both times: tracing output shows that Homer's account now contains about 600 dollars. Why didn't the *Account* class catch this?

Of course, *Account* doesn't perform any parameter checking. Neither the constructor, nor the *Withdraw*, *Deposit*, or *Transfer* methods check whether their argument values are consistent when executed. To remedy this, we could simply insert parameter checks into every method, which would however result in code duplication. The problem can be solved much more naturally in a declarative way—by specifying the intention of parameter checking rather than the algorithm of doing so.

Aspects can help us here; we will write an aspect allowing us to amend the parameter declarations of *Account's* methods with declarative attributes, specifying rules for valid values. We will use *NotNull* to indicate a parameter which must not be null, *GreaterThanZero* to indicate that a parameter must be positive, and *NotEmptyString* to indicate that a non-null string must hold a value. We will also apply these attributes to property declarations in order to check the property's value when it's being set.

Then, we will write the respective *ParameterCheckingAspect*, which checks that the rules attached to the parameters are met prior to every method execution or property setter. In contrast to the previous tracing aspect, this aspect should not be configured at assembly level; the concern of parameter checking is vital for the behavior of the *Account* class and should therefore stay near *Account's* definition. We will thus configure the aspect directly on the class with a class-level configuration attribute. Since the aspect will not hold any state, we can use a *SingletonAspectAttribute*.

The modified *Account* class and the attribute definitions are given in listing 4.43. The listing also declares a common interface for all validation attributes, which will make it easier to handle them in an aspect.

For writing the aspect, we need to consider advice for execution of constructors and ordinary methods as well as for property setters. We can handle constructors and methods in the same way: we simply iterate through their parameter lists and check each parameter for violated rules—the corresponding advice is shown in listing 4.44.

```
[SingletonAspect(typeof(ParameterCheckingAspect))]
public class Account {
  private string id;
  private decimal balance;
  private string owner;

  public Account([NotNull,NotEmptyString]string id,
      [GreaterThanZero]decimal initialBalance,
      [NotNull,NotEmptyString]string owner) {
    this.id = id;
    this.balance = initialBalance;
    this.owner = owner;
  }

  public virtual string ID {
    get { return id; }
  }

  public virtual decimal Balance {
    get { return balance; }
    protected set { balance = value; }
  }

  [NotNull] public virtual string Owner {
    get { return owner; }
    set { owner = value; }
  }

  public virtual void Deposit([GreaterThanZero]decimal amount) {
    this.Balance += amount;
  }

  public virtual void Withdraw([GreaterThanZero]decimal amount) {
    this.Balance -= amount;
  }

  public virtual void Transfer([GreaterThanZero]decimal amount,
      [NotNull]Account to) {
    to.Deposit(amount);
    this.Withdraw(amount);
  }

  public override string ToString() { /* as before */ }
}

public interface IParameterChecker {
  bool IsValid(ParameterInfo parameter, object value);
}

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Property)]
public class GreaterThanZeroAttribute : Attribute, IParameterChecker {
  public bool IsValid(ParameterInfo parameter, object value) {
    object zero = Activator.CreateInstance(parameter.ParameterType);
    return ((IComparable)value).CompareTo(zero) > 0;
  }
}

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Property)]
public class NotNullAttribute : Attribute, IParameterChecker {
  public bool IsValid(ParameterInfo parameter, object value) {
    return value != null;
  }
}

[AttributeUsage(AttributeTargets.Parameter | AttributeTargets.Property)]
public class NotEmptyStringAttribute : Attribute, IParameterChecker {
  public bool IsValid(ParameterInfo parameter, object value) {
    return !object.Equals(value, string.Empty);
  }
}
```

Listing 4.43: *Acocunt* class with declarative parameter checking

```
[Advice(typeof(BeforeMethod))]
[Advice(typeof(BeforeConstruction))]
[Priority(-10)]
public void CheckParameters(MethodBase methodOrConstructor, object[] args) {
  ParameterCollection parameters =
    AspectEnvironment.GetParameterCollection(methodOrConstructor, args);
  for (int i = 0; i < parameters.Count; ++i) {
    IEnumerable<IParameterChecker> checkers =
      parameters[i].GetCustomAttributes<IParameterChecker>();
    CheckParameter(parameters[i], checkers);
  }
}

private static void CheckParameter(ParameterCollection.Parameter parameter,
  IEnumerable<IParameterChecker> checkers) {
  foreach (IParameterChecker checker in checkers) {
    if (!checker.IsValid(parameter.Info, parameter.Value)) {
      throw new ArgumentException(checker.GetType().Name + " not fulfilled.",
        parameter.Name);
    }
  }
}
```

Listing 4.44: Advice for method and constructor parameter checking

The advice is bound to both *before method* and *before construction* join points—this is possible because we only access context information which is available for both join point kinds. We give the advice a priority of -10, i.e. slightly above the default priority of 0—this ensures that by default this advice is executed prior to any other (e.g. prior to the method tracing advice). The advice uses the *ParameterCollection* introspection mechanism in order to enumerate the parameters of the method or constructor and to retrieve the validation rules declared on them.

For parameter checking in property setters, we need a second advice, which needs to extract the validation attributes from the property definition itself (rather than the method), and we have to validate them against the single *value* parameter of the setter method, as shown in listing 4.45. The advice is also bound to *BeforeMethod* join points, but only to those of setter methods of properties which have at least one *IParameterChecker* attribute defined on them.

```
[Advice(typeof(BeforeMethod))]
[WhereSetter]
[WhereDefined(typeof(IParameterChecker), Context=Context.Property)]
[Priority(-10)]
public void CheckPropertyValue(object target, MethodInfo setter, object[] args) {
  ReflectedProperty<object> property =
    AspectEnvironment.PropertyOf<object>(target, setter);
  IEnumerable<IParameterChecker> checkers =
    property.GetCustomAttributes<IParameterChecker>(true);
  ParameterCollection parameters =
    AspectEnvironment.GetParameterCollection(setter, args);
  CheckParameter(parameters["value"], checkers);
}
```

Listing 4.45: Advice for property value checking

When put together, these two advice and one helper method form the whole aspect needed in order to implement declarative parameter checking. Running the extended test driver now leads to exceptions when trying to exploit the behavior shown before.

Comparison

As already mentioned, we could have implemented the concern of parameter checking imperatively rather than declaratively—in fact, this is how it is usually done. This would have involved an *if/throw* block for each parameter to be checked at the beginning of the respective method.

As with the logging aspect, we will shortly compare the aspect-oriented solution with the traditional one.

Advantages of the Aspect-Oriented Solution

Code locality and understandability: In the aspect-oriented version, the code for parameter checking is much better separated: within the *Account* class, only short declarative attributes hint at what rules apply to the parameters; the actual rule implementation is encapsulated in dedicated attribute classes, and the implementation of parameter checking is cleanly packaged into the parameter checking aspect. Again, this improves understandability and readability of the *Account* class; the method bodies are concise and are easily analyzed and understood. In contrast, in an object-oriented version, large parts of the method bodies (with the one-line implementations of the *Deposit* and *Withdraw* methods at least 50%, but likely more) would consist of parameter checking and would distract from the actual purpose of the method.

Code size and effort: Seen in isolation, the methods in the *Account* class are much shorter in the aspect-oriented solution than they would be in an object-oriented implementation. But even when considering the aspect code, it's much less effort to add parameter checking in the aspect-oriented implementation: the parameter checking code, e.g. for ensuring that amounts need to be greater than zero, is reused instead of retyped; only the (short) declarative attribute needs to be duplicated.

Declarativity for documentation: The declarative validation rules allow to grasp the preconditions for each parameter simply by looking at its declaration, without needing to read the method body. The attributes therefore act as documentation as well as program source code, increasing understandability even more.

Reusability of the aspect: The parameter checking aspect is highly reusable and can be readily applied to any other target class.

Scalability of the aspect: Parameter checking can easily be applied to any parameter of any method with a single attribute specification.

Changeability of the class: The functional semantics (i.e. the method bodies) of the *Account* class can easily be changed without affecting the parameter checking.

Changeability of the aspect: Parameter checking semantics (e.g. changing the exception to be thrown if a parameter is null) can easily be changed in one single place without affecting the *Account* class.

Disadvantages of the Aspect-Oriented Solution

***ObjectFactory* and virtual methods:** Again, all instances of *Account* must adhere to the design restrictions implied by XL-AOF. In particular, objects not created with the *ObjectFactory* will not check their method parameters.

4.4.4 Concern 3: Call Privileges Aspect

Another way how Homer's account could be outbalanced in the original test driver would be that of *impersonating* Jack and transferring money away from Jack's account. Consider again an extended test driver implementing this in listing 4.46.

```
Account homer = ObjectFactory.Create<Account>("10242083", 1000m, "Homer");
homer.Deposit(100.0m);
homer.Withdraw(100.0m);

Account jack = ObjectFactory.Create<Account>("213456757", 2000m, "Jack");
homer.Transfer(1532.95m, jack);

TryOut(delegate {
  jack.Transfer(600m, homer);
});
```

Listing 4.46: Transferring back the money

Implementation-wise, there is no reason why this shouldn't work—if Homer can get an instance of Jack's account, he can also call its *Transfer* method. In order to avoid this unwanted situation, a security model is needed. Whenever an *Account* object is created (or, in a more realistic scenario, loaded from a storage system), the respective creator should be required to authenticate. And while everyone is allowed to deposit money and transfer money to an acocunt, only the owner is allowed to withdraw or transfer money away from it. And this is our next cross-cutting concern.

Using aspects, it's simple to implement privileges checking: we will create a per-object aspect which retrieves a security token from the user when an *Account* object is created. And whenever a method is executed, *before method* advice checks if the user is permitted to execute the operation.

In order to declaratively configure the required privileges, we can define a custom attribute *OwnerOnlyAttribute*, which is applied to the *Withdraw* method of the account class—the action which can only be performed by the owner. Since *Transfer* calls that operation, it is automatically guarded as well. *Deposit* is not tagged and is thus still available to anyone using an *Account* object.

Listing 4.47 shows the *Account* class with the new *Withdraw* method as well as the custom attribute declaration. Like the parameter checking aspect, the security aspect is applied directly onto the class (using a *PerObjectAspectAttribute*)—this really depends on whether the security aspect is seen to be inherent to the *Account* class or if a use of *Account* without security checks is conceivable. In our case, security is vital, therefore the aspect is tightly integrated.

```
[SingletonAspect(typeof(ParameterCheckingAspect))]
[PerObjectAspect(typeof(SecurityAspect))]
public class Account {
  ... // unchanged

  public virtual void Deposit([GreaterThanZero]decimal amount) {
    this.Balance += amount;
  }

  [OwnerOnly]
  public virtual void Withdraw([GreaterThanZero]decimal amount) {
    this.Balance -= amount;
  }

  // [OwnerOnly] implicit with call to Withdraw
  public virtual void Transfer([GreaterThanZero]decimal amount, [NotNull]Account
    to) {
    to.Deposit(amount);
    this.Withdraw(amount);
  }

  ... // unchanged
}

...

[AttributeUsage(AttributeTargets.Method)]
public class OwnerOnlyAttribute : Attribute { }
```

Listing 4.47: Adding an *owner only* security restriction to the *Account* class

For our demonstration purposes, we will implement the security aspect in a very simple way—we need an *after construction* advice to gain a security token and a *before method* advice to check the token as explained above, but in the actual implementation, we will not employ any sophisticated authentication system.

Instead, the security token is generated when an *Account* is constructed simply by the user entering his name via the console. When a method tagged with the *OwnerOnly* attribute is executed, we check whether this name matches the string of the *Account's owner* field. This aspect is shown in listing 4.48.

```
public class SecurityAspect {
  private string user;
  private ReflectedField<string> owner = null;

  [Advice(typeof(AfterConstruction))]
  public void DoLogin(object instance) {
    Console.Write("Please login for using account '" + instance + "': ");
    user = Console.ReadLine();
  }

  [Advice(typeof(BeforeMethod))]
  [WhereDefined(typeof(OwnerOnlyAttribute))]
  public void CheckForOwnerPrivileges(MethodInfo action) {
    if (user != owner.Value) {
      throw new SecurityException(action.Name + " can only be performed by " +
        owner.Value + ".");
    }
  }
}
```

Listing 4.48: Aspect ensuring that the *OwnerOnlyAttribute* is respected

While this is a really simple implementation, integrating this with Windows authentication services, for example, would not require much more work on the aspect side.

Comparison

Without an aspect-oriented approach, we could have implemented this concern directly within the *Account* class, having the user authenticate in the constructor and checking for privileges at the beginning of the *Withdraw* method. So, what are the benefits of doing it the aspect-oriented way?

Advantages of the Aspect-Oriented Solution

Code locality and understandability: Implementing the security concern inline within the *Account* method wouldn't really be much of a problem, as there is only one method to protect. However, the aspect-oriented solution still has the benefit of better encapsulation of concerns—adding authentication to the constructor, security checks to *Withdraw*, and a *user* field to the *Account* class wouldn't make the class easier to understand, on the contrary. Even with this simple situation, the aspect-oriented solution makes for a better readable and more easily understandable solution.

Declarativity for documentation: One can immediately see whether everyone or only the owner of an account can perform an operation simply by looking at the method declaration, without parsing the code; the declarative approach here again serves as a means of documentation. In the object-oriented version, this would have to be explicitly documented.

Reusability of the aspect: While only supporting two different kinds of permissions ("owner-only" and "everyone"), the aspect can be applied to every class where this concept is needed—as long as the target class has an *owner* field.

Scalability of the aspect: Security checks can easily be applied to any new method with a single declarative attribute.

Changeability of the class: When changing the functional concerns of the *Account* class, the security aspect is not affected.

Changeability of the aspect: When changing the security concern, the *Account* class is not affected.

Disadvantages of the Aspect-Oriented Solution

***ObjectFactory* and virtual methods:** As with the previous aspects, the restrictions imposed by XL-AOF make for a disadvantage of the aspect-oriented solution. (Since these restrictions are inherent to XL-AOF, we will apply this disadvantage to every aspect in these tutorials.)

Code size and effort: In this case, the effort is actually a little higher with the aspect-oriented solution since we did not achieve any code reuse in our example. The situation would however turn around as soon as additional classes or methods to be protected are added to the program.

4.4.5 Concern 4: Atomic Operation Aspect

Incorporating the security check into the program makes another problem apparent. Take a look at the excerpt of the tracing output generated by the previous section's extended test driver shown in listing 4.49, paying attention to how the balance of Homer's account changes.

```
...
Please login for using account 'Jack's account, current balance: USD 2.000,00':
  Homer
...
STARTING: Transfer, target: Jack's account, current balance: USD 3.532,95
STARTING: Deposit, target: Homer's account, current balance: USD -532,95
FINISHED: Deposit, target: Homer's account, current balance: USD 67,05
STARTING: Withdraw, target: Jack's account, current balance: USD 3.532,95
FAILED: Transfer: Withdraw can only be performed by Jack.
ERROR: Withdraw can only be performed by Jack.
Homer's account, current balance: USD 67,05
```

Listing 4.49: Tracing output generated by the previous test driver

The security aspect correctly aborts the *Transfer* operation as soon as the invalid *Withdraw* operation is started, however, it doesn't roll back the changes (i.e. the *Deposit* operation) made up to this moment!

While we could easily work around this by either turning around the statements in the *Transfer* method or by applying the *OwnerOnlyAttribute* to the *Transfer* method as well as to *Withdraw*, this wouldn't solve the fundamental problem of *Transfer* having to be an atomic operation—if any of its suboperations fail at any time, the whole operation needs to be rolled back.

This atomicity requirement makes for an interesting cross-cutting concern. It will be implemented with an aspect more complex than those presented so far.

We will start by defining how we can achieve atomicity. Although it's a simplification, we will define for this sample that at the beginning of an atomic operation the states of all objects involved in the operation (this means the target object itself as well as all of the method's parameters) need to be saved. If (and only if) an exception is thrown in the course of the operation, the states need to be restored before the exception leaves the method.

Actually, this by itself is easily formulated as an aspect with an *around method* advice, as shown in listing 4.50. However, the atomicity concern depends on an additional cross-cutting concern: storeability. The *CreateSnapshot* and *snapshot.Restore* methods need a way to store and load the state of the target object.

We now have two possibilities—either make use of the .NET-integrated serialization and demand that *Account* be declared *Serializable*, or implement our own serialization mechanism. The first possibility is much cleaner, using an already available and tested mechanism, and should therefore definitely be preferred whenever possible. We will however choose the second one for the sake of explaining how to implement a serialization aspect with XL-AOF: we implement a *StoreabilityAspect*, which the *AtomicityAspect* depends on.

For this example, we will only perform a very simple sort of serialization: we will store the values of all fields of the target object in a dictionary object and, on deserialization, reassign the values from the dictionary to the fields. Of course,

```
[AttributeUsage(AttributeTargets.Method)]
public class AtomicAttribute : Attribute { }
...
public class AtomicityAspect {
  [Advice(typeof(AroundMethod))]
  [WhereDefined(typeof(AtomicAttribute))]
  public object AroundMethod(object target, object[] args, IMethodProceeder
    nextProceeder) {
    Snapshot snapshot = CreateSnapshot(target, args);
    try {
      return nextProceeder.Proceed(args);
    }
    catch {
      snapshot.Restore();
      throw;
    }
  }

  private Snapshot CreateSnapshot(object target, object[] args) {
    ... // shown later
  }
}

class Snapshot {
  ... // shown later
}
```

Listing 4.50: Aspect implementing atomicity for arbitrary methods, incomplete

a real serialization mechanism would have to analyze the fields' types and act accordingly, but for the account example, it will suffice. A more sophisticated serialization aspect for space-based computing will be given in the next chapter.

To implement the serialization mechanism, we will first define an *IStoreable* interface, as shown in listing 4.51. Objects implementing this interface provide *SaveState* and *RestoreState* methods for storing and loading their internal states into and from dictionary objects.

```
public interface IStoreable {
  void SaveState(Dictionary<string, object> state);
  void RestoreState(Dictionary<string, object> state);
}
```

Listing 4.51: *IStoreable* interface for storing and restoring object state

Of course, the *Account* class does not implement this interface; therefore, the *StoreabilityAspect* needs to add this interface via introduction—the respective *StoreableIntroducer* nested class shown in listing 4.52 implements the interface for the target objects.

The *StoreableIntrocer* class gets access to the target object's fields via an injected *ReflectedFieldCollection*. Its implementations of *SaveState* and *RestoreState* can therefore easily iterate through the fields and store/restore their contents in/from a dictionary object. Storing only makes sense for value types and (immutable) string objects—for other objects only a reference rather than the content would be stored. For reference types, the serialization mechanism needs to be recursively invoked; and the aspect does so for objects which themselves implement *IStoreable* (either directly or via an aspect).

```
public class StoreabilityAspect {
  [Introduce]
  public class StoreableIntroducer : IStoreable {
    ReflectedFieldCollection<object> targetFields =
      ReflectedFieldCollection<object>.Injected;

    public void SaveState(Dictionary<string, object> state) {
      foreach (ReflectedField<object> field in targetFields) {
        if (field.FieldType.IsValueType || field.FieldType == typeof(string)) {
          state[field.Name] = field.Value;
        }
        else if (field.Value is IStoreable) {
          Dictionary<string, object> innerState = new Dictionary<string,
            object>();
          ((IStoreable)field.Value).SaveState(innerState);
          state[field.Name] = innerState;
        }
        else {
          throw new InvalidOperationException("Can only automatically implement
            IStoreable on classes holding " +
            "only primitive types, value types, and IStoreables as fields.");
        }
      }
    }

    public void RestoreState(Dictionary<string, object> state) {
      foreach (ReflectedField<object> field in targetFields) {
        if (field.FieldType.IsValueType || field.FieldType == typeof(string)) {
          field.Value = state[field.Name];
        }
        else if (field.Value is IStoreable) {
          Dictionary<string, object> innerState = (Dictionary<string,
            object>)state[field.Name];
          ((IStoreable)field.Value).RestoreState(innerState);
        }
        else {
          throw new InvalidOperationException("Can only automatically implement
            IStoreable on classes holding " +
            "only primitive types, value types, and IStoreables as fields.");
        }
      }
    }
  }
}
```

Listing 4.52: Using introduction to implement *IStoreable*

As the last step in implementing the atomicity aspects, we fill in the missing pieces of the *AtomicityAspect*, now that we have the *StoreabilityAspect*, as shown in listing 4.53. The *AtomicityAspect* depends on *StoreabilityAspect*. Due to the *RequiresSingletonAspect* attribute, XL-AOF will automatically add the storeability aspect to every target class the atomicity aspect is applied to (unless the user has manually and explicitly applied the storeability aspect).

The listing now also shows the implementation of the *Snapshot* utility class, which simply acts as a wrapper for a number of state dictionaries, and the *CreateSnapshot* method, which stores the target object itself and any parameters implementing the *IStoreable* interface.

```
[AttributeUsage(AttributeTargets.Method)]
public class AtomicAttribute : Attribute { }

[RequiresSingletonAspect(typeof(StoreabilityAspect))]
public class AtomicityAspect {
  [Advice(typeof(AroundMethod))]
  [WhereDefined(typeof(AtomicAttribute))]
  public object AroundMethod(object target, object[] args, IMethodProceeder
    nextProceeder) {
    Snapshot snapshot = CreateSnapshot(target, args);
    try {
      return nextProceeder.Proceed(args);
    }
    catch {
      snapshot.Restore();
      throw;
    }
  }

  private Snapshot CreateSnapshot(object target, object[] args) {
    Snapshot ss = new Snapshot();
    ss.TakeSnapshot((IStoreable)target);
    for (int i = 0; i < args.Length; ++i) {
      ss.TakeSnapshot(args[i] as IStoreable);
    }
    return ss;
  }
}

class Snapshot {
  private Dictionary<IStoreable, Dictionary<string, object>> snapshots = new
    Dictionary<IStoreable, Dictionary<string, object>>();

  public void TakeSnapshot(IStoreable o) {
    if (o != null) {
      Dictionary<string, object> state = new Dictionary<string, object>();
      o.SaveState(state);
      snapshots.Add(o, state);
    }
  }

  public void RestoreSnapshot(IStoreable o) {
    if (o != null) {
      o.RestoreState(snapshots[o]);
    }
  }

  public void Restore() {
    foreach (IStoreable snappable in snapshots.Keys) {
      RestoreSnapshot(snappable);
    }
  }
}
```

Listing 4.53: Complete atomicity aspect

Listing 4.54 finally shows the adapted *Account* class implementing an atomic *Transfer* method. Again, we use class-level aspect configuration to express that this aspect is inherent to the *Account* class, that it cannot exist without it. And indeed, running the test driver with the new aspect attached to the class works as expected—the method fails, and the *Deposit* operation is rolled back.

Comparison

Now, in this particular situation, we could have implemented the rollback of the *Transfer* method simply by storing the account balances in local variables

```
[SingletonAspect(typeof(ParameterCheckingAspect))]
[PerObjectAspect(typeof(SecurityAspect))]
[SingletonAspect(typeof(AtomicityAspect))]
public class Account {
  ... // unchanged

  [Atomic]
  public virtual void Transfer([GreaterThanZero]decimal amount, [NotNull]Account
    to) {
    to.Deposit(amount);
    this.Withdraw(amount);
  }

  ... // unchanged
}
```

Listing 4.54: *Account* class extended to support atomic operations

at the beginning of the method and by reassigning them in *catch* blocks when an exception is caught. Is the aspect-oriented solution, which is clearly more complex, really any better?

Advantages of the Aspect-Oriented Solution

Code locality and understandability: Once again, the aspect-oriented solution improves code locality. Instead of tangling the *Transfer* method with exception handling and roll-back code, we managed to implement that code outside of the class. While this needed more work, since we wanted make to the aspect generally reusable, it's worth it what regards readability and understandability of the code.

Declarativity for documentation: Also again, the use of a custom attribute to mark atomic operations makes for a great implicit means of documentation, enabling a reader to immediately see what operations are atomic without needing to read the full code.

Reusability of the aspect: Because the serialization and atomicity mechanisms were implemented in a very generic way (they don't access the *balance* field by its name, for example), the aspect can be readily applied to any other target class without effort.

Scalability of the aspect: One declarative attribute is enough to add atomicity to any new method.

Changeability of the class: When changing the functional concerns of the *Account* class, the atomicity concern is not affected.

Changeability of the aspect: When changing the atomicity concern, the *Account* class is not affected.

Disadvantages of the Aspect-Oriented Solution

***ObjectFactory* and virtual methods:** As always, the design restrictions imposed by XL-AOF apply.

Code size and effort: And again, in this situation our effort is actually much higher to achieve a generic aspect-oriented solution than to implement an ad-hoc object-oriented one. This only pays off in scenarios where we can reuse the aspect's functionality.

4.4.6 Concern 5: Property Change Notification Aspect

To conclude this section about XL-AOF, we will demonstrate the implementation of one last concern. Suppose we want to show the *Account* class in a Windows Forms application, employing data binding to display account data in a window as shown in figure 4.2.

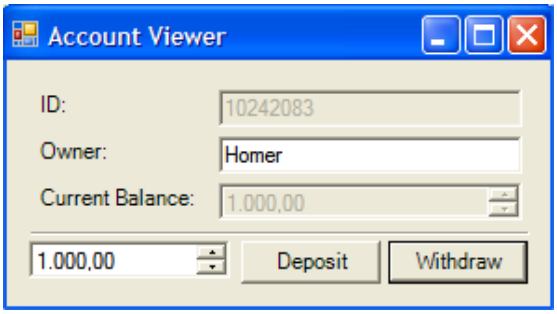

Figure 4.2: Account viewing application

Creating such a window is simple with Windows Forms, binding the account data to the different controls is even simpler and requires only three lines of code thanks to Windows Forms Data Binding, as demonstrated in listing 4.55. However, running the application and using either the withdraw or dispose functionality shows that something is missing: Windows Forms Data Binding is not notified when the account's balance changes, and the shown values are therefore not updated.

```
id.DataBindings.Add("Text", account, "ID");
owner.DataBindings.Add("Text", account, "Owner");
balance.DataBindings.Add("Value", account, "Balance");
```

Listing 4.55: Using Data Binding in order to display the account's properties

To make data binding work correctly, we would need to implement the *INotifyPropertyChanged* interface which comes with Windows Forms. Its implementors need to provide an event called *PropertyChanged*, which, when invoked, tells Windows Forms Data Binding which property has changed, so the display can be updated.

Property change notifications for Windows Forms are a cross-cutting concern, therefore we will use an aspect to implement them. We use introduction to add the interface to the *Account* class, and we create an *around advice* for all property setters, which invokes the introduced event when the property's value changes. Listing 4.56 shows the full implementation of aspect and introducer.

```
public class NotificationAspect {
  [Introduce]
  private PropertyChangedIntroducer propertyChangedIntroducer = new
    PropertyChangedIntroducer();

  [Advice(typeof(AroundMethod))]
  [WhereSetter]
  public object AroundPropertySet(object target, MethodInfo method, object[] args,
     IMethodProceeder nextProceeder) {
    ReflectedProperty<object> property =
      AspectEnvironment.PropertyOf<object>(target, method);
    object oldValue = property.InvokeGetMethod();
    object returnValue = nextProceeder.Proceed(args);
    object newValue = property.InvokeGetMethod();
    if (!object.Equals(oldValue, newValue)) {
      propertyChangedIntroducer.Raise(target, property.Name);
    }
    return returnValue;
  }

  class PropertyChangedIntroducer : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;

    public void Raise(object sender, string propertyName) {
      if (PropertyChanged != null) {
        PropertyChangedEventArgs args = new
          PropertyChangedEventArgs(propertyName);
        PropertyChanged(sender, args);
      }
    }
  }
}
```

Listing 4.56: Full change notification aspect

The aspect's around advice simply extracts the property's value before and after assignment, and, if the values differ, instructs the introducer to raise the notification event. The introducer itself is perfectly simple—it only adds the event and provides a method to raise it. We use a field as a means for introduction rather than a nested class; this way, the introducer can be accessed from the advice method.

Change notification is not a concern inherently linked to the *Account* class. Instead, it's only necessary for data binding; therefore we will use a global configuration attribute rather than a target class-level one. Listing 4.57 shows the configuration, the *Account* class itself has not changed.

```
[assembly: GlobalSingletonAspect(typeof(Account), typeof(NotificationAspect))]
```

Listing 4.57: Configuring the notification aspect

Comparison

With the change notification aspect, we have implemented a very frequent concern: every class to be used with Windows Forms Data Binding, and these are numerous, needs to implement the *INotifyPropertyChanged* interface. The most important strength of the aspect is its reusability—once written, data binding gets a breeze to be implemented on any class.

An object-oriented solution for just the *Account* would have been simple, however. Implementing the interface just adds one event to the class, and the event

needs to be raised from only two properties. Once again, we will therefore compare the two ways of solving this concern.

Advantages of the Aspect-Oriented Solution

Code locality and understandability: As could be seen from all aspects implemented in these tutorial sections, AOP implementations of cross-cutting concerns tend to improve code locality and understandability by way of better separation of concerns. Unless used in a wrong or unsuitable way, this is a constant benefit of the aspect-oriented paradigm and applies also in this case, where all change notification concern code is cleanly modularized in an aspect.

Reusability of the aspect: As already indicated, reusability is the most important asset of this aspect. It can be used to make virtually any class behave correctly with data binding with no changes being made to the class itself.

Reusability of the class: The *Account* class can easily be reused without notification.

Scalability of the aspect: The aspect automatically includes any new property added to the *Account* class.

Changeability of the class: When changing the functional concerns of the *Account* class, the notification concern is not affected.

Changeability of the aspect: When changing the notification aspect, the *Account* class is not affected.

Disadvantages of the Aspect-Oriented Solution

***ObjectFactory* and virtual methods:** As always, the design restrictions imposed by XL-AOF apply.

Code size and effort: With this aspect, we are already saving code duplication, because there are two properties which would need to be instrumented with change notification code in an object-oriented implementation, whereas we only write the aspect code once. Still, the effort spent for write the aspect is higher than simply implementing the concern within the *Account* class. Once the aspect exists, however, it is a real code saver.

Chapter 5

AO-DCL—An Aspect-Oriented Distributed Concern Library

After having described a lightweight and extensible aspect-oriented framework in the previous chapter, we will now return to the core motivation of our work: providing an efficient means for space-based application development. In this chapter, we present an AO-DCL, an *aspect-oriented distributed concern library*, which provides a number of pre-implemented concerns common to distributed computing.

The AO-DCL will be based on two grounds: XL-AOF, the framework introduced in the previous chapter, will be the aspect-oriented and declarative infrastructure we build the concerns on, and *.NET &Co* [Tec04c] will provide the interface to the (CORSO) space needed for implementing the concerns with. To the client, most of the imperative .NET &Co-related details will be hidden by the aspects, since we aim at providing a declarative, goal-oriented rather than an imperative, algorithmic concern library.

We will divide the concerns we are dealing with in two groups. First, we will present a number of *low-level* concerns dealing with the basic requirements of space-based applications—sharing of objects, data representation, notifications, and suchlike. After these, we will present a selection of common *high-level* requirements, such as caching, monitoring, and error handling concerns, which are implemented with the low-level aspects previously shown. Depending on the concrete application scenario, programmers can either directly employ those high-level aspects fitting their needs or use them as an impulse for implementing their own. The latter is even more important than providing highly reusable generalized space aspects, because many space-based applications comprise distributional concerns tightly coupled to the application scenario. For that reason, we will give source code listings and class diagrams for these aspects wherever necessary.

We will not show complete implementations of the concerns in this chapter, especially not of the low-level concerns. In parts, the implementations are al-

ready available from our previous publications (most notably "Attributes &Co—Collaborative Applications with Declarative Shared Objects" [eKS05a]), and all others are trivially implemented with the information given in this chapter. We will also not do extensive performance benchmarks of the aspect-oriented concerns: most aspects are completely equivalent with their object-oriented implementations (within the limits of a proxy-based infrastructure we presented in chapter 3); if not, we will include a corresponding remark in the aspect's description. Also, reusability of the aspects presented in this chapter is prioritized very high (higher than ideal performance, for example), because the purpose of our distributed concern library is to provide solutions for many distributed scenarios, not just a few specialized ones.

For the low-level concerns, we will present each aspect using the following structure:

Goal and usage scenario: This section briefly describes the concern's goals, giving its motivation and communicating its underlying use cases.

Classic realization: We first show the problems or shortcomings of implementing the concern with the classic .NET API, which should be remedied by the aspect.

Aspect-oriented realization: Then, we show how to implement the concern using XL-AOF.

- **Design:** We start by introducing the aspect design, giving the involved entities and roles (aspects, introducers, target objects, etc.), and their relationships.
- **Implementation:** Then, we explain what features of XL-AOF need to be used in order to implement the aspect. As already mentioned, we won't give full source code, but we will explain the implementation sufficiently detailed to make writing the actual code a trivial task.
- **Example usage:** Last, we give a short example of using the aspect in code.

Evaluation: After describing the realization of the concern as an aspect, we will provide a thorough evaluation of the concern implementation, analyzing whether it actually contributes to our goal of making space-based development more efficient.

Relevance for space-based computing: The concerns presented in this thesis have been designed and implemented for use in conjunction with the CORSO middleware as a space-based abstraction layer. However, CORSO is not the only implementation of space-based computing; therefore, we will shortly discuss the relevance of each concern for space-based computing in general and CORSO's successor XVSM in particular.

Because these make the heart of the distributed concern library, we will evaluate the aspects using a template adapted from "Design Pattern Implementation in Java and AspectJ" by Jan Hannemann and Gregor Kiczales [HK02] and "Foundations of AOP for J2EE Development" by Renaud Pawlak et al (pages

150ff) [PSR05], and extended by us with a few additional criteria. The template contains software quality criteria which provide a measurement for efficiency with regards to the software development process. It consists of the following items:

Code locality: As one of the most important goals of aspect-oriented programming is to achieve better separation of concerns, *code locality* evaluates the modularization of cross-cutting concerns and their separation from business code: is all the code for each specific concern modularized in a dedicated class or aspect (or a number thereof)? This naturally has direct effects on readability and maintainability of the application, which affect the efficiency of the overall development process [May99].

Understandability: When looking at the code, can one grasp without thorough reading what it is about? This criterion is about choosing appropriate names and declarative attributes, which should act as documentation and goals at the same time, which again improves readability. It is especially important for efficiency in software teams, as programmers can faster understand a piece of code created by their peers if this criterion is fulfilled.

Effective code size and effort: Given the implemented concern, how much code and effort is needed to apply it to a target class? Does it make space-based application development more efficient with regards to the time spent for actual programming?

Performance: As stated before, we will not give performance benchmarks here, but we will ask the question of whether there are any considerable performance advantages or drawbacks of the aspect-oriented implementation.

Changeability of target code: Can the target code be changed without impacts on the concern code, and can the target code be recompiled without the concern having to? This is important for maintainability and extensibility of the application, vital factors for software development efficiency [May99].

Changeability of concern code: Can the concern code change without adverse impacts on the target code, and can the concern code be recompiled without the target code having to? Again, this is important for maintainability and extensibility.

Reusability of concern code: Is the concern implementation general enough to be reused in many scenarios? What assumptions about the target code does it make? As indicated previously, this is a most vital point, since a concern library with hardly reusable concern implementation couldn't do much to improve space-based application development.

Transparency of composition: Can the concern be transparently composed with others, or does the programmer need to pay extra attention for doing so? Similar as with reusability, it is important that a concern implementation doesn't create too many dependencies or even potential sources for

errors. Otherwise, it could actually make application development less efficient.

Each aspect will first be evaluated in textual form according to these criteria. Then, we will give an evaluation matrix, grading both the aspect-oriented and the classic implementation with one of the following marks in order to give an impression of the improvements brought by the aspect at a single glance: 1 (catastrophic), 2 (bad), 3 (unsatisfactory), 4 (good), 5 (excellent).

The high-level aspects following the low-level ones will also be evaluated according to that schema. However, as they leverage the features of space-based computing based on the low-level DCL core, their presentation will therefore not include a classic implementation and their evaluation will not include a comparison with such an implementation.

5.1 General Common Infrastructure

Before presenting any aspects, we will provide some general object-oriented extensions to .NET &Co, which are needed by many of the following concerns.

5.1.1 Connection Management

Usually, space-based applications connect to one, typically local, CORSO kernel as their entry point to the shared space. Of course, .NET &Co provides support for establishing and handling connections, but it does not offer centralized access to an established connection. Many of the aspects presented later require an established CORSO connection in order to work with the space, retrieve and create API objects, etc. We therefore introduce a central *ConnectionManager* class to the API, which acts as a singleton wrapper around the *CorsoConnection* API class, instantiating and opening the CORSO connection at first use, keeping it alive while the application is running, and closing it only at application shutdown (or when it is explicitly closed).

The connection manager's interface is presented in listing 5.1, its implementation can be any of the .NET-based versions of the singleton pattern [Ske06]; in the listing, we use a static class. The parameters needed for the connection (user name, password, domain, application ID, site, and port of the CORSO kernel) can be configured either via the application's *app.config* or *machine.config* files [Ric06], as illustrated in listing 5.2, or by setting the connection manager's static properties *before* the connection is established—changes made to a connection parameter while a connection is open will lead to an error. The class also manages two kinds of timeouts which can be used consistently throughout the application: a *local timeout*, which is used to access the local CORSO kernel, and a *network timeout*, which is used to access remote CORSO kernels over the network.

The connection manager can be used to temporarily close a connection, in which case it will be reestablished the next time it is accessed. Any CORSO API objects become invalid by closing and reopening a connection, which might

lead to unforeseen errors with some of the aspects presented below. Explicit disconnects should therefore only be used in controlled situations, and in most scenarios they shouldn't be necessary.

```
public static class ConnectionManager {
  public static string Site {
    get { ... }
    set { ... }
  }

  public static int Port {
    get { ... }
    set { ... }
  }

  public static string UserName {
    get { ... }
    set { ... }
  }

  public static string Password {
    get { ... }
    set { ... }
  }

  public static string Domain {
    get { ... }
    set { ... }
  }

  public static int ApplicationID {
    get { ... }
    set { ... }
  }

  public static TimeSpan LocalTimeout {
    get { ... }
    set { ... }
  }

  public static TimeSpan NetworkTimeout {
    get { ... }
    set { ... }
  }

  public static CorsoConnection GetConnection() {
    ...
  }

  public static void CloseConnection() {
    ...
  }
}
```

Listing 5.1: Connection manager extension of .NET &Co

5.1.2 Shared Object Identity

In space-based application, their are typically objects which are shared over the space, and such which aren't. Shared objects are distinguished because they have an associated data object in the space—an object identity (*OID*)—and methods for synchronizing the object with the space.

With .NET &Co, every class can decide on its own how to implement this contract of OID and synchronization. However, for the purpose of the concern

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="corso" type="Corso.Util.Configuration.ConfigurationHandler,
      Corso.Util"/>
  </configSections>

  <corso>
    <connection>
      <site>localhost</site>
      <port>5006</port>
      <username>fabian</username>
      <!-- password set in code, empty domain, default application ID -->
      <localtimeout>-1</localtimeout> <!-- infinite timeout for localhost -->
      <networktimeout>5000</networktimeout> <!-- five seconds timeout for the
        network -->
    </connection>
  </corso>
</configuration>
```

Listing 5.2: Configuring the connection using the *app.config* file

library, we need to standardize this contract. We therefore define an interface *ISpaceObject*, as shown in listing 5.3, which defines a property for accessing the object's OID, and *Refresh* and *Persist* methods for synchronizing the object with the space. Both methods take the respective *CorsoTransaction* object to use, and the *Refresh* method also requires a timeout in case a remote kernel must be contacted for reading the object. This interface should be implemented by every shared object used in conjunction with the concern library.

```
public interface ISpaceObject {
  CorsoVarOid Oid { get; set; }
  void Refresh(CorsoTransaction tx, TimeSpan timeout);
  void Refresh(TimeSpan timeout); // reads without a transaction
  void Persist(CorsoTransaction tx);
  void Persist(TimeSpan timeout); // uses implicit transaction
}
```

Listing 5.3: Common *ISpaceObject* interface contract

5.1.3 Asynchronous Change Notifications

One of the most important space-based features for coordinating distributed processes is the possibility of subscribing to near real-time change notifications: processes registered for specific OIDs will immediately (i.e. as fast as possible, in a best-effort manner) be notified when the respective OID's data changes.

Unfortunately, .NET &Co only provides low-level support for these change events; in particular, it does not offer out-of-the-box support for asynchronous notifications. Instead, the respective API call (*CorsoNotification.Start*) blocks the calling thread until either the specified timeout expires or a notification is received.

Extending this infrastructure for the purpose of easier implementation of asynchronously watchable objects, we define a wrapper *AsyncNotificationManager* around *CorsoNotification*. The wrapper, whose interface is given in listing 5.4, provides functionality for registering and unregistering arbitrary OIDs for asynchronous notification. Its implementation simply incorporates a background

thread, which periodically call's the notficiation's *Start* method in a loop, performing the actual waiting. When a notification occurs, a *NotificationReceived* event is fired on a .NET thread pool worker thread. Subscribers to this event receive a reference to the manager and to the changed OID as well as the data sent with the notification. The manager can be disposed of, which frees any resources held by the manager and finishes the background thread.

```
public class AsyncNotificationManager : IDisposable {
  public delegate void NotificationReceivedHandler(AsyncNotificationManager
    sender, CorsoOid oid, CorsoData data);
  public event NotificationReceivedHandler NotificationReceived;

  public CorsoNotification Notification {
    get { ... }
  }

  public void Dispose() {
    ...
  }

  public void Add(CorsoOid oid) {
    ...
  }

  public void Remove(CorsoOid oid) {
    ...
  }
}
```

Listing 5.4: *AsyncNotificationManager* class

5.2 Low-Level Concerns

5.2.1 Object Serialization

Goal and Usage Scenario With every space-based application, the data to be shared via the space must be specified. Usually, there are objects within the application (business objects, e.g. account data, graphical objects, etc.), which directly map to space data items; in other words: their state needs to be serialized and deserialized into and from the space. CORSO's .NET &Co provides an interface for this purpose *(CorsoShareable*, see appendix A) which defines *Read* and *Write* methods for serializing state into a space data object.

The object serialization aspect's goal is to automate the process of implementing this interface, i.e. to make an arbitrary business class automatically serializable to the CORSO space. The implementation should provide mechanisms for declaratively configuring which fields of the class should or shouldn't be serialized and for specifying custom serialization strategies for special cases.

Problems of Platform-Independent Serialization A naive implementation of serialization could simply be based on the binary or SOAP serialization mechanisms already available on the .NET platform; serializing an object graph into a binary or string-based stream and writing that binary or string data into the space. However, this would make for extremely poor integration with other platforms: only platforms supporting the .NET serialization formats could

take part in the distributed application. Java or native C++ applications, for example, would be excluded (unless a complex parser for these formats were implemented).

Therefore, it is extremely important to make use of the space-based architecture's platform-independent interface definition language: with CORSO, this means explicit mapping of objects to *structures* and fields to either nested structures or primitive values such as integers, floating point numbers, strings, or byte arrays. For good interoperability, it's also important to have a defined serialization order for an object's fields (e.g. alphabetical order).

Implementation of such a platform-independent serialization mechanism must also provide a way to identify the .NET type associated with a space structure. While it would be possible to simply use CORSO's structure name tag to contain the full type name, this could once again hinder interoperability. It would therefore be better to have a generalized mapping mechanism, which should allow any structure name to be registered with any .NET type.

Classic Realization Classically, each business class implements the *Read* and *Write* methods defined by the *CorsoShareable* interface, calling serialization methods of the *CorsoData* object passed to the method for each field, thus serializing the object in a structured manner. Depending on the field type, *CorsoData* provides serialization methods for integer values, floating-point numbers, strings, etc.

An exemplary implementation of this concern is given in listing 5.5, where a *RectangleShape* class, which holds data representing a rectangular graphical object, is given the ability to be serialized to the space. To achieve compatibility with other clients to the space, the rectangle's members are serialized in alphabetic order: first the *bounds* field is handled by serializing its x and y coordinates, width, and height (again in alphabetic order) within a substructure of the data item, then the color field is serialized after being converted to an ARGB integer value. Deserialization involves reading the serialized data in the same order as it was written and additionally checks for invalid data.

```
public class RectangleShape : IShape, CorsoShareable {
  private Color color;
  private Rectangle bounds;

  public RectangleShape(Color color, Rectangle bounds) {
    this.color = color;
    this.bounds = bounds;
  }

  public Color Color { get { return color; } }
  public Rectangle Bounds { get { return bounds; } }

  public virtual void Draw(Graphics g) { ... }

  public void Write(CorsoData data) {
    data.PutStructTag("RectangleShape", 2); // name and arity of structure
    data.PutStructTag("Rectangle", 4); // bounds field as nested structure
    data.PutInt(bounds.Height); // fields of Rectangle in alphabetical order
    data.PutInt(bounds.Width);
    data.PutInt(bounds.X);
    data.PutInt(bounds.Y);
    data.PutInt(color.ToArgb()); // color converted to integer
  }
```

```
  public void Read(CorsoData data) {
    // check name and arity of structure
    StringBuilder structName = new StringBuilder();
    int arity = data.GetStructTag(structName);
    if (arity != 2 || structName.ToString() != "RectangleShape") {
      throw new InvalidOperationException(...);
    }

    // bounds field as a nested structure
    structName = new StringBuilder();
    arity = data.GetStructTag();
    if (arity != 4 || structName.ToString() != "Rectangle") {
      throw new InvalidOperationException(...);
    }
    int height = data.GetInt();
    int width = data.GetInt();
    int x = data.GetInt();
    int y = data.GetInt();
    bounds = new Rectangle(x, y, width, height);

    // color converted from integer
    color = Color.FromArgb(data.GetInt());
  }
}
```

Listing 5.5: Serializing *RectangleShape* object using .NET &Co

Aspect-Oriented Realization

Design Figure 5.1 shows the entities making up the aspect-oriented implementation of this concern. First, there is the role of the target class—any .NET object whose fields are to be serialized. To it, the serialization aspect is applied; we define a dedicated singleton factory attribute *SpaceSerializableAttribute* for it to improve readability of the resulting code.

At the heart of the aspect is, of course, an introduction—after all, the goal is to add an implementation of the *CorsoShareable* interface to the aspect's target class. An introducer is used to implement the *Read* and *Write* methods by accessing the target object's fields via field injection. For the actual serialization, we use a separate subsystem, represented in the figure by the *SpaceSerializerManager*.

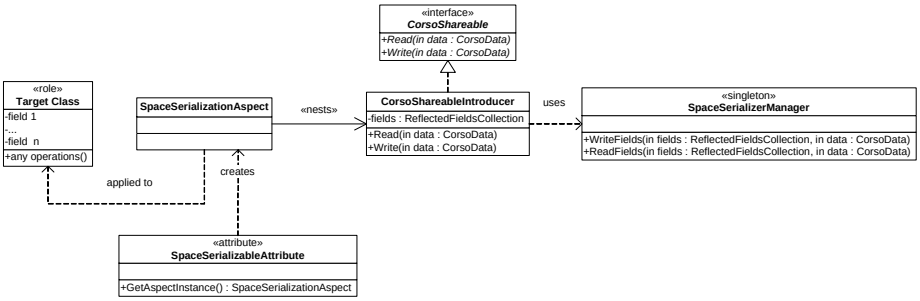

Figure 5.1: Design of the serialization aspect

The serialization subsystem itself is detailed in figure 5.2, although it's not strictly part of the aspect—in fact, it's an extension of an automatic CORSO

serializer implemented by Andreas Rottmann at the Vienna University of Technology [Rot05] in the course of our work. Its entry point is the *SpaceSerializerManager*, which provides a way of serializing a collection of fields as a structured CORSO data object by delegating to a number of serializers it manages. Each serializer has a method to read an object from the space data—either into an existing target object or into a newly allocated one, depending on whether it is a reference type or value type instance—and a method to write it into a *CorsoData* object.

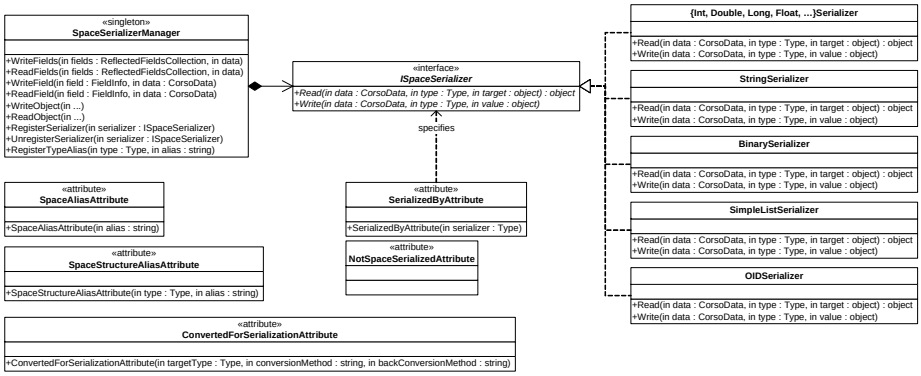

Figure 5.2: Serialization subsystem

Predefined serializers already exist for primitive types, strings, binary byte arrays, and collections which implement the *IList* interface, and new ones are easily added by implementing the *ISpaceSerializer* interface and registering the new serializer with the *SpaceSerializationManager*. If an object should be serialized whose type is not handled by a registered serializer, the manager creates a substructure and recursively serializes the fields of the object, similar to how the *bounds* member was serialized as a substructure in listing 5.5. It identifies the object's type by using the full class name as an identifier. Alternatively, it also allows to register aliases for types either imperatively by its *RegisterTypeAlias* method, declaratively by applying a *SpaceAliasAttribute* on the respective class definition, or by using an assembly-level *SpaceStructureAliasAttribute*. Whenever the serialization must instantiate a specific type while reading data from the space, it requires a parameterless constructor to be present.

The *SpaceSerializationManager* also detects when a field references an implementation of *ISpaceObject* and serializes a named structure containing only a reference to its OID instead of the whole object by using the predefined *OIDSerializer*.

If the automatic serializer selection mechanism, which selects the serializer based on the respective field type, isn't sufficient, a programmer can declaratively configure a field to be serialized with a specific serializer via the *SerializedByAttribute*. If a field is marked with the *NotSpaceSerializedAttribute*, it will be ignored by the serialization manager.

If a field needs to be converted to another type prior to serialization or deserialization, the *ConvertedForSerializationAttribute* can be used: it takes the

target type, the name of a conversion method, and the name of a back-conversion method. The conversion method must either be an instance method of the object to be converted from or a static method of the target type, the back-conversion method must be a static method of the field's static type or an instance method on the target type. This mechanism can be used to serialize types unknown to the serialization structure without reflecting over their members if only they can be converted into types known by the serializers.

Implementation The serialization aspect is easily implemented by defining the introducer as a nested class of the aspect and tagging it with an *IntroduceAttribute*. The target object's fields can be retrieved by using an injected *ReflectedFieldsCollection*.

The serialization subsystem doesn't need any aspect-oriented features, the only XL-AOF mechanism it uses is the *ReflectedFieldsCollection* it needs to enumerate the fields to be serialized.

Extensibility We allow other aspects to influence the serialization process of objects by triggering custom *before object serialization* and *before object deserialization* join point on the respective object when it is about to be serialized or deserialized. By advising these join points, other aspects can exclude specific data from the serialization process or add additional data to be serialized in addition to the object's fields. They can even avoid the whole deserialization process by returning a non-null value from the *before object deserialization* join point, in which case that value is assumed to be the deserialized object.

We also include *before field serialization* and *before field deserialization* join points triggered on the target object before serialization and deserialization of each of its fields. This allows for other aspects to extend the serialization process of single fields, giving them the ability to manipulate both the value to be serialized and the choice of serializer before any actual serialization or deserialization is performed. Aspects can avoid deserialization of a single field by returning a non-null value from the *BeforeDeserialization* join point, which is then assumed to be the deserialized value.

The custom join point definitions are given in listing 5.6.

```
public delegate void BeforeObjectSerialization(object target, CorsoData corsoData,
   ICollection<ReflectedField> fields);
public delegate object BeforeObjectDeserialization(Type targetType, CorsoData
  corsoData, ICollection<ReflectedField> fields);

public delegate void BeforeFieldSerialization(object target, CorsoData corsoData,
  ReflectedField<object> field, Type type, ref object data, ref ISpaceSerializer
  serializer);
public delegate object BeforeFieldDeserialization(object target, CorsoData
  corsoData, ReflectedField<object> field, Type type, ref ISpaceSerializer
  serializer);
```

Listing 5.6: Custom join points triggered by the serialization manager

In addition to the delegate definitions, join point handlers are defined and registered as explained in section 4.3.3 of the previous chapter.

Example Usage Listing 5.7 shows the same *RectangleShape* class as before, but now implemented using the serialization aspect. The *bounds* member is automatically serialized as a substructure named "Rectangle" due to the alias configuration, and the shape's color is declaratively configured to be converted to and from integer by specifying the respective conversion methods of the *Color* structure.

```
[assembly:SpaceStructureAlias(typeof(System.Drawing.Rectangle), "Rectangle")]

[SpaceSerializable]
public class RectangleShape : IShape {
  [ConvertedForSerialization(typeof(int), "ToArgb", "FromArgb")]
  private Color color;
  private Rectangle bounds;

  public RectangleShape(Color color, Rectangle bounds) {
    this.color = color;
    this.bounds = bounds;
  }

  public Color Color { get { return color; } }
  public Rectangle Bounds { get { return bounds; } }

  public virtual void Draw(Graphics g) {
    g.DrawRectangle(Bounds, Color);
  }
}
```

Listing 5.7: Making *RectangleShape* serializable using an aspect

Evaluation

Code locality (CL): The classic version mixes a cross-cutting concern—the implementation of the *CorsoShareable* interface—into the code of the target class, which makes for bad code locality. Neither serialization code nor target code are cleanly encapsulated as dedicated classes, tangling occurs on class-level (although not on method level). The aspect-oriented version separates serialization and target code; both are cleanly encapsulated in modularized entities.

Understandability (U): To understand that a class is serializable in the classic version, one only needs to look at the list of implemented interfaces. To fully understand what fields are serialized and whether special serialization code is used for them, one needs to look both at the field declarations and at the method implementations. In contrast, the aspect-oriented version also clearly states the serializability (by means of the dedicated factory attribute), but in addition allows to understand the full serialization semantics only by looking the the field declarations.

Effective code size and effort (CSE): Object-oriented implementations of the *CorsoShareable* interface generally tend to result in a lot of code lines, since it adds methods with at least two lines per serialized field. In the example, the interface implementation actually makes up most of the class body. Applying the aspect is much more effortless and results in much less code—it's just one line for the factory attribute. Additional tagging of non-serialized or special fields only needs one line for a vast minority of fields. (In the example, only one field needed to be tagged.)

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Classic	2	3	1	5	1	1	1	5	2.375 (bad)
Aspect-oriented	5	5	5	5	5	4	5	5	4.875 (good)

Table 5.1: Evaluation matrix for serialization aspect

Performance implications (PI): Since the fields can be accessed using fast field access (see 3.2), no significant performance implications are caused by the aspect-oriented implementation.

Changeability of target code (CT): In the classic implementation, changes to the target code cannot be conducted independently from the target code, since they are implemented within the same class. In the aspect-oriented version, target code can easily be changed without the aspect having to be adapted, separate recompilation is possible.

Changeability of concern code (CC): In the classic implementation, concern code cannot be changed independently from the concern code, since they are implemented within the same class. In the aspect-oriented version, concern code can easily be changed without the target code having to be adapted as long as none of the per-field attributes changes its name or interface. Separate recompilation is possible.

Reusability of concern code (RC): In the classic implementation, concern code cannot be reused, serialization code needs to be repeated (with modifications) for every new target class. In the aspect-oriented implementation, the aspect is general enough to be reused on any target class.

Transparency of composition (TC): The concern has no implications on other concerns in both implementations.

Relevance for Space-Based Computing Of course, the described serialization mechanism depends on CORSO, other space-based systems however have similar cross-cutting serialization mechanisms which can be encapsulated as an aspect. The interface names and the details of the interface definition language provided by the space-based abstraction differ, but a similar mechanism is also used for serializing objects into XVSM containers and entries.

5.2.2 Shared Object Identity

Goal and Usage Scenario Earlier in this chapter, we have established a common contract for objects with a shared identity with the *ISpaceObject* interface. The goal of this concern is to associate a .NET class with a shared identity by implementing the *ISpaceObject* interface on the class. This involves backing the class with an internal OID variable of type *CorsoVarOid*[1] and implementing the interface's methods and properties by delegating to the OID object. It does not involve deciding when exactly a shared object is to be refreshed or persisted, which is a concern dealt with later in this chapter.

[1]We ignore CORSO's feature of constant objects for this explanation. A corresponding aspect could be implemented in much the same way as this one is.

Classic Realization Implementation of the *ISpaceObject* always follows the same process: first, a variable of type *CorsoVarOid* is added to the respective class; then, the *Refresh* and *Persist* methods are implemented by delegating to the OID's *ReadShareable* and *WriteShareable* methods. The OID is usually passed to the constructor of the class, having either been newly created or retrieved from the space by the caller.

Listing 5.8 shows an exemplary implementation of the interface for the *RectangleShape* object already used to illustrate the previous concern examples. The implementation depends on the class implementing *CorsoShareable*—code for this has already been shown.

```
public class RectangleShape : IShape, CorsoShareable, ISpaceObject {
  private CorsoVarOid oid;

  private Color color;
  private Rectangle bounds;

  public RectangleShape(CorsoVarOid oid, Color color, Rectangle bounds) {
    this.oid = oid;
    this.color = color;
    this.bounds = bounds;
  }

  public Color Color { get { return color; } }
  public Rectangle Bounds { get { return bounds; } }

  public virtual void Draw(Graphics g) {
    g.DrawRectangle(Bounds, Color);
  }

  public void Write(CorsoData data) {
    // see previous concern
  }

  public void Read(CorsoData data) {
    // see previous concern
  }

  // ISpaceObject implementation

  public CorsoVarOid Oid {
    get { return oid; }
    set { oid = value; }
  }

  public void Refresh(CorsoTransaction tx, TimeSpan timeout) {
    oid.ReadShareable(this, tx, timeout);
  }

  // reads without a transaction
  public void Refresh(TimeSpan timeout) {
    oid.ReadShareable(this, null, timeout);
  }

  public void Persist(CorsoTransaction tx) {
    oid.WriteShareable(this, tx);
  }

  // uses implicit transaction
  public void Persist(TimeSpan timeout) {
    oid.WriteShareable(this, timeout);
  }
}
```

Listing 5.8: Giving *RectangleShape* object a shared identity using .NET &Co

Aspect-Oriented Realization

Design Realization of the shared object identity concern as an aspect is quite straight-forward: a space identity aspect introduces the *ISpaceObject* interface to the target object, as illustrated in figure 5.3. Again, we add a dedicated factory attribute to improve readability: the aspect is instantiated per object, because it holds object-associated state (the OID). Although not visible in the figure, the aspect requires the presence of the serialization aspect (or another implementation of the *CorsoShareable* interface) presented previously.

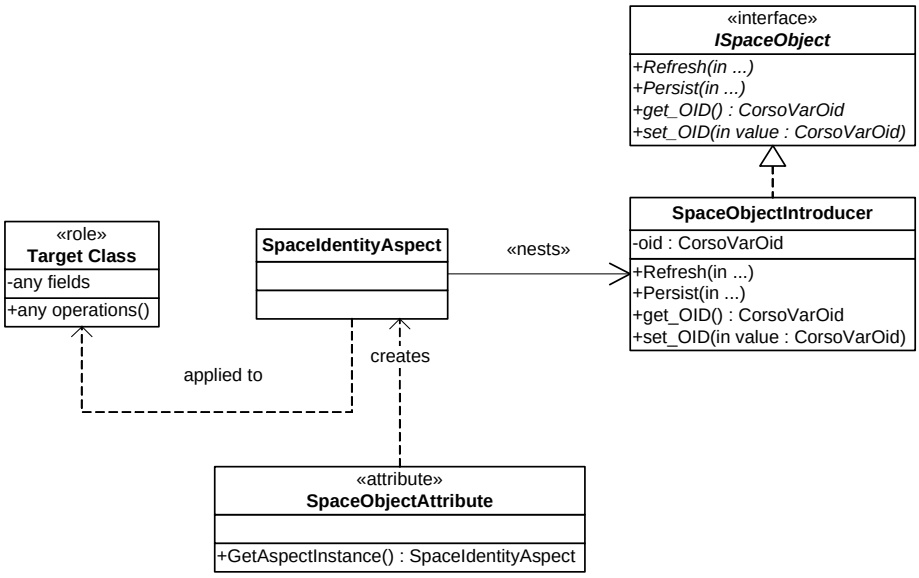

Figure 5.3: Design of the space-based identity aspect

The only remaining question is how and when an OID is to be assigned to the object. Although we could simply have the caller pass it through the constructor, as in the classic solution, this would require the instantiating code to pass more arguments to the constructor than declared in the target class—such a hidden requirement would clearly break modularization and encapsulation. We therefore define that with this aspect, the OID is unset (*null*) unless explicitly set from the outside. We will present an automated way of doing so with the next concern.

Implementation Implementation of this concern as an aspect follows its design: the aspect defines a nested class introducing the interface, which also holds the OID variable. The introduced *Refresh* and *Persist* methods delegate to the OID's *ReadShareable* and *WriteShareable* methods, as with the classic realization. The dependency on the serialization aspect is expressed via XL-AOF's *RequiresAspectAttribute*.

Example Usage Listing 5.9 shows an example usage of this aspect. The only trace of the concern in this code is the application of the factory attribute to the *RectangleShape* class. The serialization aspect from the previous section is automatically added to the class through the aspect dependency.

```
[assembly:SpaceStructureAlias(typeof(System.Drawing.Rectangle), "Rectangle")]

[SpaceObject]
public class RectangleShape : IShape {
  [ConvertedForSerialization(typeof(int), "ToArgb", "FromArgb")]
  private Color color;
  private Rectangle bounds;

  public RectangleShape(Color color, Rectangle bounds) {
    this.color = color;
    this.bounds = bounds;
  }

  public Color Color { get { return color; } }
  public Rectangle Bounds { get { return bounds; } }

  public virtual void Draw(Graphics g) {
    g.DrawRectangle(Bounds, Color);
  }
}
```

Listing 5.9: Giving *RectangleShape* object a shared identity using an aspect

Evaluation

Code locality (CL): With the classic implementation, the cross-cutting concern of space identity is not cleanly separated from the functional concerns (e.g. managing a rectangular graphical shape), tangling occurs at class and constructor level (no other methods are involved); this mixture makes for bad code locality. The aspect-oriented solution, on the other hand, separates identity and functional concerns; for both concerns, code is located in separate modules.

Understandability (U): In the classic version, it can easily be understood whether a class has a space-based identity by looking at its list of implemented interfaces. To grasp the details of this identity, however, the constructor must be mentally untangled, and the methods and properties added by the interface must be located. In the aspect-oriented version, the former can also be understood by looking at the list of attributes applied to the class. In addition, however, the aspect also improves understandability of the details because it enforces consistency: the details are the same for every target class the aspect is applied to. While this consistency is also possible with a classic implementation, it is not enforced and can therefore not be relied on.

Effective code size and effort (CSE): Classic implementation of this concern requires effort for adding a constantly high amount of code to each class (one variable, one line in the constructor, methods and properties of the interface). In the aspect-oriented implementation, exactly one line of code (the factory attribute) is added to each class definition.

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Classic	2	3	2	5	2	2	2	4	2.75 (bad)
Aspect-oriented	5	5	5	5	5	5	5	5	5 (excellent)

Table 5.2: Evaluation matrix for space-based identity aspect

Performance implications (PI): The classic and aspect-oriented versions do not differ in performance, as interface introduction performs more or less the same as explicit interface introduction does.

Changeability of target code (CT): In the classic implementation, changes to the target code can only be conducted independently from the identity concern in a limited way, since the concerns are tangled at constructor level. In the aspect-oriented version, target code can change without affecting the concern code, separate recompilation is possible.

Changeability of concern code (CC): The classic version only allows the identity concern to be changed independently from the target code in a limited, because they are located in the same class and tangling occurs at constructor level. In the aspect-oriented version, the aspect can change without affecting the target code, separate recompilation is possible.

Reusability of concern code (RC): In the classic version, the identity concern is not reusable, its code has to be repeated for each class. Although it would be possible to implement parts of the concerns in a base class, this would make for severe design restrictions. The aspect, on the other hand, is highly reusable and general enough to be used on any target class.

Transparency of composition (TC): The identity concern has no implications on other concerns in both implementations. In the classic implementation, the concern requires an implementation of *CorsoShareable* to be present, the aspect-oriented version automatically adds the respective aspect.

Relevance for Space-Based Computing Shared identity is a very CORSO-specific concern, because other space-based systems either do not provide the concept of uniquely identifiable data objects within the space at all or differentiate between identifiable and non-identifiable data items. The original tuple space definition works with unidentifiable tuples only, which makes the space merely a container for data items without providing the possibility of creating distributed coordination data structures. With XVSM, on the other hand, there are two kinds of objects in the space: uniquely identifiable and publishable containers, which in turn contain unidentifiable data structures called entries. Entries would be implemented simply by making them serializable, as discussed with the previous aspect. Containers, on the other hand, act similar to collections, which often makes their usage a functional rather than a cross-cutting concern. In those cases, where objects should however be mapped to non-collection containers, this section's solution works well also for XVSM.

5.2.3 OID Retrieval

As indicated in the previous section, it is often not trivial to associate a .NET object with a space OID. In principle, there are the following common scenarios:

1. Create a completely new OID for every new .NET instance of a class (and usually persist the object immediately after construction).
2. Create and register a completely new named OID, with either a constant name (given in code or *app.config* file), or a dynamically calculated name (e.g. from user input). Usually, the object should be persisted immediately after construction.
3. Retrieve the OID from a name registered at some CORSO kernel. Again, the name can be constant or dynamically calculated. The kernel to retrieve the OID from is either the local kernel managed by the *ConnectionManager* class, or it is located at an address configured via *app.config* or dynamically calculated (e.g. based on user input). Usually, the object should be refreshed immediately after construction.
4. Combine 2 and 3, depending on whether an OID of this name already exists.
5. Retrieve the OID from a shared data structure in the process of deserializing a space data item.

Scenario 1 could easily be automated by an aspect using an *after construction* advice, and scenario 5 is already handled by the serialization structure and the *OIDSerializer* class used by the serialization aspect. Automation of scenarios 2, 3, and 4 in a declarative and aspect-oriented fashion, however, cannot be done in a general way, especially if name or kernel address are dynamically calculated at runtime. These values usually come from within the functional components of the application (e.g. from within a GUI control), and they must be passed to the concern implementation in some way.

Therefore, we suggest to implement this concern not as an aspect, but to make it imperatively explicit: we provide a space object factory as an extension of the object factory provided by XL-AOF. Programmers can use this new factory to create objects with a space-based identity, and in turn get the ability to specify names and kernel addresses as necessary.

Because this concern is not implemented as an aspect, this section does not contain any comparisons or an evaluation matrix.

Realization

Design The new space object factory will act as an entry point to retrieving and creating space objects, similar to how the connection manager already provides an entry point for managing CORSO kernel connections. As the act of space object retrieval is always associated with a kernel connection, we extend

the connection manager class to provide a reference to a single *SpaceObjectFactory* instance, as illustrated in figure 5.4. For brevity, the rest of this chapter will just refer to "the *SpaceObjectFactory*" when referring to the factory referenced by the connection manager.

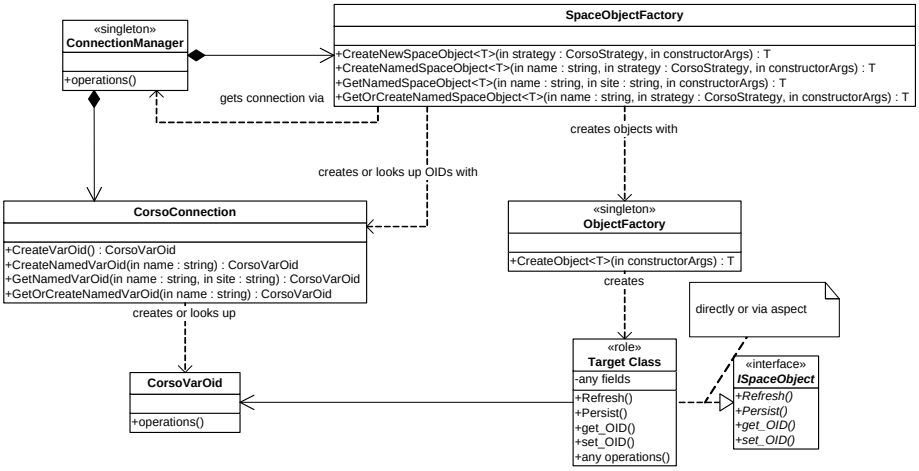

Figure 5.4: Design of the space object factory

The new factory depends on the *CorsoConnection* instance supplied by the connection manager to lookup and create OIDs, and it uses XL-AOF's *ObjectFactory* to handle the actual object instance creation. Its public interface is therefore a mixture of *CorsoConnection's* and *ObjectFactory's* public interfaces, containing generic methods for creating local instances of specific types with arbitrary constructor argument lists and associate them with named or unnamed space-based data objects. All methods which create new space objects allow the user to specify the replication strategy and persistence flags as CORSO supports eager and lazy replication as well as different persistence levels to tackle different networking scenarios and synchronicity and reliability requirements.

Implementation The factory is almost trivially implemented. Its methods first create new object instances by delegating to *ObjectFactory.Create<>*, passing it the type to be instantiated and any available constructor arguments. Then, it uses the CORSO connection obtained from the connection manager to create or retrieve an OID and assigns that OID to the created object (which must implement *ISpaceObject* either directly or e.g. via an aspect). The only point of some complexity is to determine whether the object should be refreshed or persisted after it has been created.

For this, we make use of CORSO's ability to test OIDs for values: if the OID of an object already has a value, we read it; if not, we write it instead. This ensures that the object is best-effort synchronized with the space immediately after creation. (Of course, to guarantee synchronization in a subsequent operation, transactions have to be used, as will be explained later in this chapter.)

This concern does not implement any pooling: when two .NET objects are

assigned equal OIDs (e.g. because the same name was specified for retrieving them), two separate object instances are created. This behavior was chosen for better consistency (the factory behaves more like the .NET *new* instruction that way), and it actually facilitates space-based multi-threaded programming, because threads don't have to manually synchronize access to objects with space-based identity, but can instead rely on the space concepts of transactions and notifications to achieve consistency.

To facilitate easy implementation of pooling on top of this aspect, the factory passes the OID as context information when delegating to *ObjectFactory.Create<>*. This results in any *ObjectCreation* advice getting the context information in their *additionalContext* parameter, so aspects can implement pooling based on that information.

Relevance for Space-Based Computing The idea of named space objects was introduced by CORSO, and its successor XVSM extends this by providing extended ways of publishing and looking up containers. A similar factory approach like the one just described would therefore work well for XVSM.

5.2.4 Asynchronous Change Notifications

Goal and Usage Scenario Since notifications are such an important space-based feature for coordinating distributed processes, the goal of this concern is to augment a .NET object having an associated space-based identity with an event indicating that the respective space data item has changed. Users of the object can then easily subscribe to this event and react on the changes asynchronously. This is an especially important concern for coordination and synchronization of distributed processes.

Before talking about the concrete realization of the concern, we will again provide a common contract for objects with an active notification, similar as we did with the space-based identity concern. Listing 5.10 shows the definition of an interface *IWatchedSpaceObject* which should be implemented by all objects with space-identity that provide the possibility of subscribing to change events.

```
public interface IWatchedSpaceObject {
  void Start(AsyncNotificationManager manager);
  void Stop(AsyncNotificationManager manager);

  event SpaceRepresentationChangedHandler SpaceRepresentationChanged;
}

public delegate void SpaceRepresentationChangedHandler(IWatchedSpaceObject sender,
   CorsoData newData);
```

Listing 5.10: *IWatchedSpaceObject* interface

The interface makes use of the *AsyncNotificationManager* shown earlier in this chapter: the *Start* method registers the object's OID with the notification manager, *Stop* unregisters it again, and *SpaceRepresentationChanged* fires when a notification event is received. CORSO sends the object's changed data together with the notification—this can be used to immediately deserialize the object without having to perform a *Refresh* operation.

Classic Realization Classic realization of the interface is straight-forward: the respective class is given a *SpaceRepresentationChanged* event, and *Start* and *Stop* are implemented to simply hand the object's OID over to the notification manager. In *Start*, the object needs to subscribe to the notification manager's *NotificationReceived* event: when this event is fired for the corresponding OID, the *SpaceRepresentationChanged* event must be raised as well. Listing 5.11 shows such an implementation of the interface on the *RectangleShape* object introduced in the previous examples.

```
public class RectangleShape : IShape, IWatchedSpaceObject, ISpaceObject,
  CorsoShareable {
  private CorsoVarOid oid;
  public event SpaceRepresentationChangedHandler SpaceRepresentationChanged;

  ... // code from previous and functional concerns left out

  public void Start(AsyncNotificationManager manager) {
    manager.NotificationReceived += manager_NotificationReceived;
    manager.RegisterOid(oid);
  }

  public void Stop(AsyncNotificationManager manager) {
    manager.UnregisterOid(oid);
    manager.NotificationReceived -= manager_NotificationReceived;
  }

  private void manager_NotificationReceived(AsyncNotificationManager sender,
    CorsoVarOid oid, CorsoData data) {
    if (oid.Equals(this.oid) && SpaceRepresentationChanged != null) {
      SpaceRepresentationChanged(this, data);
    }
  }
}
```

Listing 5.11: Classic implementation of the *IWatchedSpaceObject* interface

Aspect-Oriented Realization

Design Aspect-oriented realization of this concern is just as straight-forward as the classic implementation: an introducer adds the *IWatchedSpaceObject* interface to a target class, as shown in figure 5.5. The introducer needs a reference to the target object in order to access the OID (it depends on the *ISpaceObject* to be implemented by the target object) and for the *SpaceRepresentationChanged* event's parameter. For convenience and documentation purposes, a factory attribute is defined as well, which instantiates the aspect in a per-object fashion. (Per-object instantiation is needed because the introducer needs to hold a reference to the target object.)

Implementation For the interface introduction, a nested class introducer is used, which receives the target object as an injected field. To ensure that the target implements *ISpaceObject*, the aspect explicitly requires the space identity aspect. Apart from this, the interface implementation is equivalent to the classic implementation.

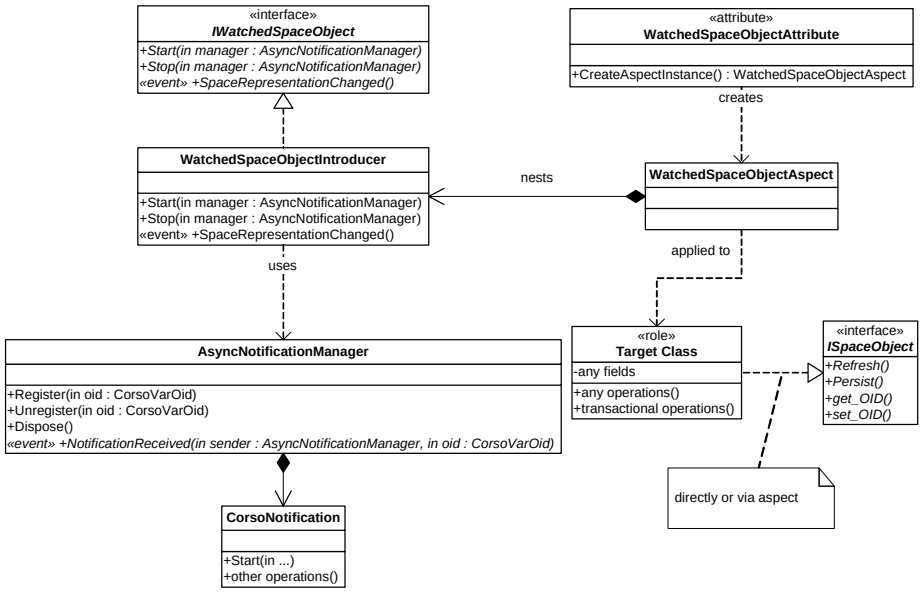

Figure 5.5: Implementing the notification concern in an aspect-oriented way

Example Usage Listing 5.12 shows the same *RectangleShape* class as before, this time made asynchronously watchable via an aspect. The space identity aspect is automatically added by the *WatchedSpaceObjectAspect's* dependencies, so it is not necessary to explicitly specify the *SpaceObjectAttribute*. Notification handling for a *RectangleShape* instance will start as soon as its new *IWatchedSpaceObject.Start* method is called, and it can be stopped via *IWatchedSpaceObject.Stop*.

```
[WatchedSpaceObject]
public class RectangleShape : IShape {
  ... // code from previous and functional concerns left out
}
```

Listing 5.12: Implementation of the *IWatchedSpaceObject* interface using an aspect

Evaluation

Code locality (CL): The classic implementation mixes the cross-cutting concern and functional concerns at class-level; this tangling makes for bad modularization, although no tangling occurs on method level. The aspect-oriented solution provides clear separation of concerns: all notification-related code is in a dedicated aspect.

Understandability (U): In the classic implementation, it is immediately visible that a class supports asynchronous notifications by looking at the list of the interfaces it implements. To understand the details of notifications, only the methods added by the interface need to be located and read,

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Classic	2	4	2	5	3	3	2	5	3.25 (unsatisf.)
Aspect-oriented	5	5	5	5	5	5	5	5	5 (excellent)

Table 5.3: Evaluation matrix for notification aspect

which makes for good understandability. Understandability of the aspect-oriented version is even better, however: not only can it be immediately determined that a class is a watchable space object by looking at its list of attributes—the factory attribute again acts as documentation and goal at the same time—, the details are also situated in a well-known place: the aspect implementation.

Effective code size and effort (CSE): The interface implementation code needs to be repeated for every watchable class, adding two methods and an event to the class. This adds unnecessary effort and makes the code longer than necessary. Because the interface implementation is readily available via introduction in the aspect-oriented realization, effort and code size are minimized: only one attribute needs to be added to the class definition.

Performance implications (PI): There are no adverse performance implications to be expected by either realization.

Changeability of target code (CT): In the classic implementation, changes to the target code can be conducted independently from the concern code, but separate recompilation is not possible. The aspect-oriented realization allows for both changes to be made and separate recompilation.

Changeability of concern code (CC): The classic implementation does allow changing the concern implementation independently from changing code dealing with functional concerns, but separate compilation is not possible. The aspect-oriented realization also allows changes to be made, and separate compilation is also supported.

Reusability of concern code (RC): In the classic implementation, concern code cannot be reused, since it is integrated into the target code. Because it does not depend on the target code, it could be refactored into a base class, however this would require all watchable objects to be derived from this class and would thus severely restrict an application's class design. In the aspect-oriented solution, the aspect can be reused for any target class.

Transparency of composition (TC): The notification concern has no implications on other aspects in either realization. In the classic solution, the concern requires an OID to be available (shared identity concern), the aspect-oriented version automatically adds the respective aspect to the target class if necessary.

Relevance for Space-Based Computing Other space-based implementations as well provide notifications, however their specifics differ according to the respective space model. With tuple spaces, it is usually possible to register

for changes on the whole space and be notified when an item with a specific structure (and maybe values) is added to the space. With XVSM, notifications are offered for container changes (e.g. items being added or removed), but also—via XVSM's interception mechanism—for ordinary entry read and write operations. Depending on how the mapping between .NET objects and XVSM containers/entries is solved, notifications could become functional rather than cross-cutting concerns. If a mapping between business objects and watchable containers is required, an aspect-oriented solution would again be superior to an object-oriented one.

5.2.5 Transactional Operations

Goal and Usage Scenario The space object factory provides best-effort synchronization at object construction time, and notifications can be used to invoke an object's *Refresh* method in near real-time when its space representation changes. Still, these mechanisms cannot safely guard against concurrency conflicts, when two or more processes modify a space object in parallel.

For this, transactional operations must be used. Before the actual operation is performed, a transaction must be created and the object (usually) be refreshed. After the operation has finished, the object must be persisted and the transaction be committed. Since CORSO uses an optimistic locking model, the transaction can fail at commit time, and usually needs to be restarted. If an error occurs while conducting a transactional operation, the transaction needs to be rolled back.

Classic Realization Classically, transactions are implemented ad-hoc for each operation: at the beginning of a transactional operation, a *CorsoTransaction* is created, then the actual operation is conducted, refreshing and persisting involved objects when needed, and, at the end, the transaction is committed. Concurrency exceptions are manually caught, and in most cases, the transaction is simply retried in case of such an error. This same process needs to be repeated for every transactional operation of every space-based object, so code is heavily repeated.

Following the previous graphical shapes examples, listing 5.13 shows an example implementation of a transactional *AddShape* operation of a *CompoundShape* container class. A compound shape simply contains a collection of subshapes, which together form one larger shape. The *AddShape* operation adds a new shape to the collection. In order to guarantee correct concurrent behavior, *AddShape* needs to be transactional, including a refresh and a persist operation ensuring synchronization with the space.

Aspect-Oriented Realization

Design Our aspect-oriented solution for this concern is based on an aspect with an around method advice bound to methods tagged with a dedicated

```
public class CompoundShape : IShape, ISpaceObject, CorsoShareable {
  private List<Shape> shapes;

  public void AddShape(Shape shape) {
    CorsoTopTransaction tx =
      ConnectionManager.GetConnection().CreateTopTransaction();
    try {
      this.Refresh(tx, ConnectionManager.NetworkTimeout);
      shapes.Add(shape);
      this.Persist(tx);
      tx.Commit(ConnectionManager.NetworkTimeout);
    }
    catch {
      tx.Abort();
      throw;
    }
  }

  // other methods left out
}
```

Listing 5.13: Classic implementation of a transactional operation

declarative *TransactionalAttribute*. Before such a method is executed, the advice acquires a transaction, conducts a refresh operation on the current object if necessary, performs the operation, persists the object (if necessary), and then disposes of the transaction. The attribute can be used to configure this process, specifying whether the object on which the method is invoked needs to be refreshed or persisted during the operation; by default, full synchronization including refresh and persist is conducted. For applying the aspect to a class, a dedicated factory attribute is provided instantiating the aspect as a singleton.

The actual transaction management is centered around a thread-affine transaction manager, which the aspect delegates to. Due to its thread affinity, it can handle nested operations: if a transactional operation calls other such operations (directly or indirectly), the same transaction instance is used for all of these. The manager also ensures that space objects are only read at most once (in order not to overwrite changes made by methods previously executed in the same transaction) and written at most once, deferring the persist operation to the end of the transaction (because CORSO does not allow objects to be written more than once per transaction).

If an exception occurs during execution of an operation, the aspect uses the manager to abort the thread's current transaction. The manager will only commit the thread's transaction instance when the first operation, the one for which it was actually instantiated, successfully finishes. If committing the transaction fails due to concurrency reasons, the aspect will attempt to repeat the operation up to a defined amount of times. By default, the aspect will continue to try conducting the operation up to ten times, then throw an exception. Programmers can change this limit by setting the *TransactionalAttribute's MaxNumberOfRetries* property to a different value. They can also request that the aspect should wait for a random amount of time before repeating the operation by setting its *MinWaitOnRetry* and *MaxWaitOnRetry* properties.

Figure 5.6 shows a UML diagram of the entities just described.

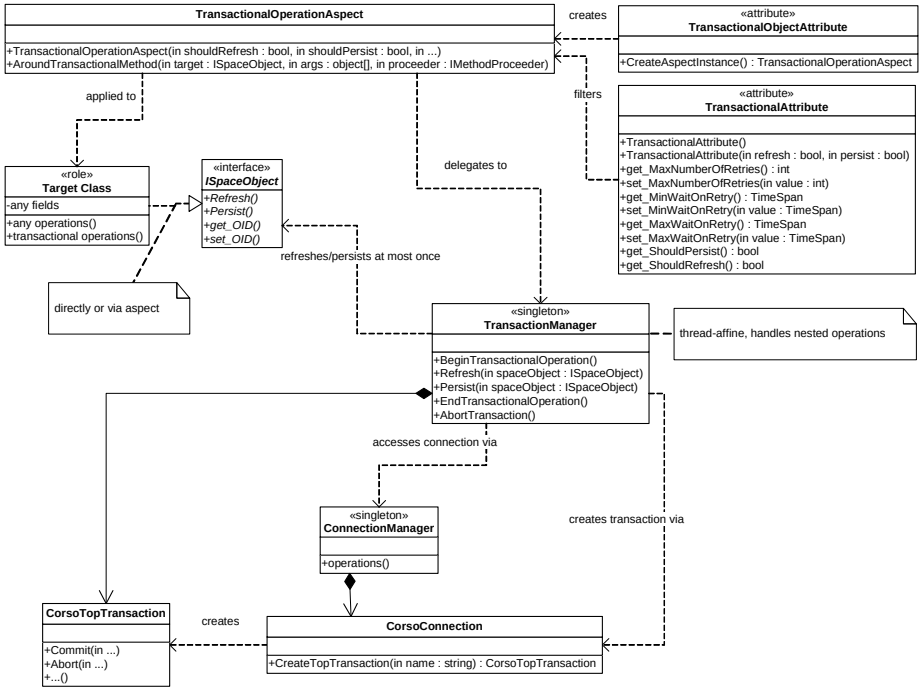

Figure 5.6: Aspect-oriented design of the transactional operations concern

Implementation Implementing the aspect itself is not much of a challenge: with the *AroundMethod* join point kind and the *WhereDefinedAttribute* filter, XL-AOF provides all which is necessary to implement the advice for intercepting transactional operations; and the aspect can immediately delegate to the transaction manager.

The transaction manager is implemented as a global singleton; however, it needs to store transactions in a thread-affine way. For this, we use .NET's *ThreadStatic* mechanism for storing thread-local data. When the *BeginTransactionalOperation* is called for the first time on a thread, the manager initializes its thread-local transaction variable, as well as a reference count. For each subsequent call to *BeginTransactionalOperation*, the reference count is increased, *EndTransactionalOperation* calls decrease the count. When it reaches zero, the transaction is actually committed and the thread-local variable is cleared again. *AbortTransaction* immediately aborts the current transaction and resets both transaction variable and reference count.

Besides the current thread's transaction, the manager also handles a thread-local set of refreshed objects (so that every object is refreshed at most once) and of objects to be persisted immediately before the transaction is to be committed (so that all objects are only persisted once at the end of the transactional operation). Objects which have already been registered for being persisted will also not be refreshed in order not to overwrite changes made by previous methods.

Concurrency exceptions getting raised when the manager commits the trans-

action are caught by the advice method, which can then decide to reconduct the operation or to rethrow the exception, depending on how the aspect was configured.

Extensibility In order to allow other aspects to handle objects engaged in a transaction in a special way, we include custom *object enters transaction* and *object leaves transaction* join points. The former is triggered on the target object just before the first transactional operation on the object is conducted in the context of a transaction, and before the object is refreshed (if necessary). The latter is triggered on the target object after the transaction an object was engaged in has been committed or aborted. The custom join point definitions are given in listing 5.14.

```
public delegate void ObjectEntersTransaction(object target, CorsoTopTransaction
  tx, MethodInfo transactionalOperation, object[] args);
public delegate void ObjectLeavesTransaction(object target, CorsoTopTransaction
  tx);
```

Listing 5.14: Custom join points triggered by the transaction aspect

Example Usage Listing 5.15 shows the same *CompoundShape* container class as in the classic implementation before, now implementing the *AddShape* operation using the transactional operation aspect.

```
[SpaceObject, TransactionalObject]
public class CompoundShape : IShape {
  private List<IShape> shapes;

  [Transactional]
  public void AddShape(Shape shape) {
    shapes.Add(shape);
  }

  // other methods left out
}
```

Listing 5.15: Implementation of a transactional operation using an aspect

Evaluation

Code locality (CL): Made obvious by the listing given in this section, code locality with the classic implementation is catastrophic: the transactional operation concern is truly crosscutting, its implementation being tangled with functional concerns not only at class level, but even at method level. Contrarily, the aspect manages to cleanly encapsulate this complex crosscutting concern, taking it out both from the method and the class.

Understandability (U): Because methods such as *AddShape* in the example tangle the cross-cutting concern with functional concerns in classic implementations, code is made very hard to understand. One needs to mentally disentangle a method body in order to understand each of the concerns. It is no coincidence that transactions are a very popular application of AOP, making the advantages of this paradigm obvious (cf. declarative

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Classic	1	1	1	5	1	1	1	4	1.875 (catastr.)
Aspect-oriented	5	5	5	4	5	5	5	4	4.75 (good)

Table 5.4: Evaluation matrix for transactional operation aspect

transactions for example in Spring [JH+05] or JBoss AOP [FR03]): an aspect-oriented implementation of this concern makes the code immediately understandable, since it removes any tangling and leaves only one concern per method.

Effective code size and effort (CSE): The classic implementation needs to repeat code (and the procedure of implementing it) for every transactional method of every space-based class. The aspect-oriented implementation, as shown in the example listing, requires only one factory attribute per class and one tag attribute per method, minimizing the amount of code necessary.

Performance implications (PI): The aspect effectively contains the same code as methods in a classic implementation would. The only additional runtime effort are the thread-local transaction bookkeeping as well as the management of the collections of refreshed and persisted objects, both of which are made necessary to make the aspect general enough to be reusable. The performance impact of these, especially when seen in the context of space-based distributed transactions, as well as refresh and persist operations, is however not significant.

Changeability of target code (CT): In the classic implementation, target code cannot be changed without affecting concern code: both are located within the same methods. In the aspect-oriented implementation, this is easily possible, including separate recompilation.

Changeability of concern code (CC): In the classic implementation, concern code cannot be changed without affecting target code: both are located within the same methods. In the aspect-oriented implementation, this is easily possible, including separate recompilation.

Reusability of concern code (RC): Since the concern code is tangled with business code in the classic implementation, it cannot be reused. In the aspect-oriented realization, the concern implementation is general enough to be applied to arbitrary classes and methods.

Transparency of composition (TC): For both implementations, the concern of transactional operations requires that all refresh and persist operations belonging to the transaction are executed between creation and committing of the transaction instance. This might create ordering dependencies when additional concerns are included.

Relevance for Space-Based Computing Clean encapsulation is a common goal for all systems providing transactional operations. Declarative transactions are implemented for example by Spring [JH+05], JBoss [FR03], and the COM+-related *System.EnterpriseServices* namespace present in the .NET framework

[Bey01]. Therefore, an aspect-oriented implementation of declarative transactions is relevant to all distributed space-based implementations with support for transactions, although the details differ with the transaction model.

5.3 High-Level Concerns

5.3.1 Space-Based Monitoring

Monitoring use cases make a large class of application scenarios for space-based computing. The data-centered space model provides a very natural mechanism for publishing diagnostic data, enabling distributed monitoring processes to read and analyze it, reacting on any problems as fast as possible. In this section, we therefore present a number of reusable aspects for making .NET applications easily debuggable and monitorable via the space.

Directories

The concerns presented in this section deal with requirements of diagnosis and monitoring. Often, these concerns will be applied to business objects whose functional concerns do not actually involve the space; in fact, the implementations we discuss are designed for full transparency. As a result, the user of these objects should not be bothered with the question of how to publish the space data items created by the concerns.

Therefore, we will first present a generic common directory data structure for the space, which can be used to publish and quickly find diagnosis objects belonging to a certain process on a certain machine. The directory can publish any object by means of the automatic serializer mechanism presented earlier in this chapter.

For each concern, there should be one instance of the directory data structure registered as a named object on the local kernel. The named object is a structure used as a linear vector of entries, which contain references to the diagnostic data objects of the concern. For identification purposes, the entries also include the machine name, process name, and process ID of the creating process, as well as the date and time of registration in the directory.

The directory class itself provides a static factory method to create the specific directory instances with predefined persistence and replication strategy values; we use lazy replication—i.e. the directory will only be replicated to other kernels on demand—and standard, non-redundant persistence. Adding, removing, retrieving, and searching directory entries is performed via transactional and thread-safe methods; for convenience, there also exist a number of helpful wrappers around these. The whole directory implementation is given in listing 5.16.

While *AddObject* is very simple, the *RemoveObject* method should be used with care, since it needs to compare its parameter to the different directory entries in order to determine which one to remove. The default way of comparison on the .NET platform—the *Equals* method—will however not always work as intended:

for reference types, it performs a reference equality check, which, due to the serialization infrastructure creating new instances on deserialization, will always yield false. Therefore, the *RemoveObject* method only works correctly for value types (whose default implementation of *Equals* should work), for reference types which explicitly implement *Equals* in a way compatible to space serialization, and for implementations of *ISpaceObject* (for which *RemoveObject* compares the OID rather than references or contents). *RemoveAt* can however be used in all scenarios.

For retrieving or enumerating the entries of the directory, *GetDirectoryEntries* can be used, which obtains a copy of the list of entries within a transaction. Built on top of it, there are a number of filtering methods for getting all the entries created by a certain machine, a certain process, or a process with a given name.

```
[SpaceSerializable]
public class DirectoryEntry<T> {
  public readonly int ProcessID;
  public readonly string ProcessName;
  public readonly string MachineName;
  public readonly DateTime CreationTime;
  public readonly T PublishedObject;

  public DirectoryEntry(T publishedObject) {
    this.PublishedObject = publishedObject;
    System.Diagnostics.Process process =
      System.Diagnostics.Process.GetCurrentProcess();
    this.ProcessID = process.Id;
    this.ProcessName = process.ProcessName;
    this.MachineName = Environment.MachineName;
    this.CreationTime = DateTime.Now;
  }
}

[TransactionalObject, WatchedSpaceObject]
public class SpaceDirectory<T> {
  public static SpaceDirectory<T> GetOrCreate(string name) {
    return SpaceObjectFactory.GetOrCreateSpaceObject<SpaceDirectory<T>>(name, new
      CorsoStrategy(CorsoStrategy.LAZY | CorsoStrategy.RELIABLE1));
  }

  private List<DirectoryEntry<T>> entries;
  private object monitor = new object(); // thread synchronization object

  [Transactional]
  public void AddDirectoryEntry(DirectoryEntry<T> entry) {
    lock (monitor) {
      entries.Add(entry);
    }
  }

  public void AddObject(T obj) {
    AddDirectoryEntry(ObjectFactory.Create<DirectoryEntry<T>>(obj));
  }

  [Transactional]
  public void RemoveObject(T obj) {
    lock (monitor) {
      // check whether obj implements ISpaceObject interface
      // if yes, its OID will be used to find the respective item
      // otherwise, the object will be checked via Equals
      ISpaceObject spaceObject = obj as ISpaceObject;
      DirectoryEntry<T> entryToBeRemoved = entries.Find(delegate (DirectoryEntry
        entry) {
        if (spaceObject != null && entr is ISpaceObject) {
          return
            ((ISpaceObject)entry.PublishedObject).Oid.Equals(spaceObject.Oid);
        }
```

```
        else {
          return entry.Equals(obj);
        }
      });
      if (entryToBeRemoved != null) {
        entries.Remove(entryToBeRemoved);
      }
    }
  }

  [Transactional]
  public void RemoveAt(int index) {
    lock (monitor) {
      entries.RemoveAt(index);
    }
  }

  [Transactional]
  public List<DirectoryEntry<T>> GetDirectoryEntries() {
    lock (monitor) {
      return new List<DirectoryEntry<T>>(entries);
    }
  }

  public DirectoryEntry<T> GetDirectoryEntry(string machineName, int processID) {
    return GetDirectoryEntries().Find(delegate (DirectoryEntry<T> entry) {
      return entry.MachineName == machineName && entry.ProcessID = processID;
    });
  }

  public IEnumerable<DirectoryEntry<T>> GetDirectoryEntriesForMachine(string
    machineName) {
    return GetDirectoryEntries().FindAll(delegate (DirectoryEntry<T> entry) {
      return entry.MachineName == machineName;
    });
  }

  public IEnumerable<DirectoryEntry<T>> GetDirectoryEntries(string machineName,
    string processName) {
    return GetDirectoryEntries().FindAll(delegate (DirectoryEntry<T> entry) {
      return entry.MachineName == machineName && entry.ProcessName = processName;
    });
  }

  public IEnumerable<DirectoryEntry<T>> GetDirectoryEntriesForProcess(string
    processName) {
    return GetDirectoryEntries().FindAll(delegate (DirectoryEntry<T> entry) {
      return entry.ProcessName = processName;
    });
  }
}
```

Listing 5.16: Directory data structure for holding diagnostic objects

Mirroring

Goal and Usage Scenario The first diagnostic concern we describe is simple, but also incredibly useful: imagine a situation where it is vital that the states of .NET business objects can be inspected at any time by an administrator or system monitoring process over a network.

To implement this idea, this concern defines space-based *mirrors* of these objects, data objects reflecting the objects' current states. Using a simple space debugger or a dedicated monitoring tool, an administrator can inspect and check a mirror's data; using notifications, one can be notified whenever the respective object is changed.

Aspect-Oriented Realization

Design Mirroring works by giving the target object a space-based identity and publishing its OID in a public directory. After execution of methods which change the object's state in an interesting way, the space-based identity is updated to reflect the new object state.

This concern is realized simply as an aspect combining *after construction* and *after method* advice with the space identity aspect in order to implement the mirroring behavior, registering the object in the public directory and persisting it as necessary.

To clean up the directory, the aspect causes the mirrored object representations to be removed from the directory when their local counterpart is claimed by the garbage collector. While this works, it is not guaranteed that all finalizers run on hard application exits, so orphaned mirror objects can be left behind. Therefore, the directory should be periodically cleared by the system administrator or monitoring system.

Implementation The public directory is simply an instance of the directory data structure described in the previous section. The concern implementation contains a class handling creation and retrieval of this instance, calling it *Mirrors*, which is used by the aspect for registering its target objects. As illustrated in listing 5.17, it adds a dependency to the space identity aspect (and thus indirectly the serialization aspect), which causes each target object to be stored as a single space object with a dedicated OID when entered into the directory.

```
public class Mirrors {
  public static readonly SpaceDirectory<ISpaceObject> Instance =
    SpaceDirectory.GetOrCreate("Mirrors");
}

[AttributeUsage(AttributeTargets.Method)]
public class UpdatesMirrorAttribute : Attribute { }

[RequiresPerObjectAspect(typeof(SpaceIdentityAspect))]
public class MirrorAspect {
  [Target]
  ISpaceObject target = ReflectedField<ISpaceObject>.InjectedTarget;

  [Advice(typeof(AfterConstruction))]
  public void AfterConstruction([AutoCast]ISpaceObject constructedObject) {
    CorsoVarOid oid = ConnectionManager.GetConnection().CreateVarOid();
    constructedObject.Oid = oid;
    constructedObject.Persist(ConnectionManager.LocalTimeout);
    Mirrors.Instance.AddObject(constructedObject);
  }

  [Advice(typeof(AfterMethod)), WhereDefined(typeof(UpdatedMirrorAttribute))]
  public void AfterMethod([AutoCast]ISpaceObject target) {
    target.Persist(ConnectionManager.NetworkTimeout);
  }

  protected override void Finalize() {
    Mirrors.Instance.RemoveObject(target);
  }
}
```

```
[AttributeUsage(AttributeTargets.Class)]
public class MirroredObjectAttribute : Attribute {
  [AspectFactory]
  public MirrorAspect CreateInstance() {
    return ObjectFactory.Create<MirrorAspect>();
  }
}
```

Listing 5.17: Aspect for mirroring object data in the space

Mirror registration is implemented using an *after construction* advice, which registers the target object immediately after its construction, creating a new OID for the mirrored object. This saves the programmer from instantiating the object with the *SpaceObjectFactory* class and thus makes mirroring completely transparent. Deregistration on finalization is implemented by having a *Finalize* method on the per-object aspect. As soon as the mirrored object becomes eligible for garbage collection, the aspect becomes so as well, so its finalize method is invoked when the the object is collected. To make the space-based mirror always reflect the latest status, the aspect comes in association with an *UpdatesMirrorAttribute* and an *after method* advice filtered by that attribute. The advice causes the mirrored object to be written to the space after each execution of a method tagged with the attribute.

To improve readability of the resulting code, the aspect comes with a dedicated factory attribute (*MirroredObjectAttribute*), which instantiates the aspect with per-object scope in order to get the finalizer to behave correctly.

Example Usage Usage of the aspect is very simple. As shown in listing 5.18, it is enough to apply the *MirroredObjectAttribute* to the business object's class definition and tag all methods which should cause the mirror to be updated with the *UpdatesMirrorAttribute*. A mirror for the object is automatically created and registered upon creation and deregistered at finalization time.

```
[MirroredObject]
public class BusinessObject {
  private string data;

  public virtual string Data {
    get { return data; }
    [UpdatesMirror] set { data = value; }
  }
}
```

Listing 5.18: Applying mirroring to any .NET business object

Evaluation

Code locality (CL): Code locality is excellent with this concern implementation: mirroring code is completely separated from the business code. Mirroring concern and data structure are cleanly modularized in separate aspects and classes.

Understandability (U): It is clearly communicated that an object has a space-based mirror through the *MirroredObjectAttribute*. Together with

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	3	5	5	5	3	4.5 (good)

Table 5.5: Evaluation matrix for mirroring aspect

the attribute used to tag methods which should cause the mirror to change, it serves as documentation and as a goal to the aspect framework at the same time.

Effective code size and effort (CSE): Making a business class mirrorable is very simple, because all imperative code is encapsulated in the aspect and the data structure classes. Only one line of code is needed, plus one line of code for methods that should cause the mirror to be updated.

Performance implications (PI): When an object is mirrored, it is to be expected that those methods which cause the mirror to be updated will need somewhat longer to execute because of the serialization and persist operations triggered by the aspect.

Changeability of target code (CT): Target code can be changed without affecting the concern code, separate recompilation is possible.

Changeability of concern code (CC): Concern code can be changed without affecting the target code, and separate recompilation is possible.

Reusability of concern code (RC): The aspect is implemented in a general way and can be applied to any target class.

Transparency of composition (TC): The aspect's dependencies automatically add a shared identity to its target objects. This might cause conflicts if a target object is also part of a different coordination data structure. However, since this concern is meant for transparent monitoring of objects not otherwise present in the space, it should not be applied in such situations.

Relevance for Space-Based Computing Distributed monitoring of objects generally is an ideal application of space-based computing, because the space concept provides a natural way of realizing it. The specific implementation of course differs with the space implementation, depending e.g. on questions such as the presence of space object identity, lookup mechanisms, and the implementation of low-level concerns.

Tracing Aspects

Goal and Usage Scenario A second important diagnostic concern is *tracing*. Applications often write a history of method calls, uncaught exceptions, and object instantiations into a local log file in order to be able to reproduce the behavior of an application in an unforeseen condition. The same mechanism can also be used to conduct program profiling [PWBK05] and usability testing [AB+03]. Writing the log to the space rather than a local disk has a few advantages, the most important one being the ability of monitoring the application over the network as it is running.

Aspect-Oriented Realization

Design To start with, the tracing concern needs a certain infrastructure: a way of just "appending" data to the space in a linear fashion, similar as one would do with a local log file. For this, we define a space data structure: a linked list, to which new items are always prepended, so that the first item is always the newest one. Similar to the mirrors of the previous concern, log lists are published in a named directory, providing immediate access to all traces of all processes connected to a certain CORSO kernel.

The tracing itself is performed by a number of aspects, one for each variety of this concern: tracing of object instantiations, method executions, and errors. Each of these comes with a dedicated factory attribute, which allows to control which kinds of traces should be generated. The object construction tracing aspect is not further configurable, but the method and exception tracing aspects come with *ExecutionTraced* and *ErrorTraced* attributes, which allow fine-grained declarative specification of what join points should actually be logged. These attributes were including because tracing *all* method calls for a class quickly results in huge logs in most situation. Therefore, it is usually better to select only a subset of calls to methods one is actually interested in. The error tracing aspect works the same way for reasons of consistency.

Implementation Using the low-level aspects, the log list data structure is easily implemented, as shown in listing 5.19. Its reverse linking structure has three advantages: firstly, any monitoring process can simply watch the top-level *LinearSpaceLog* object in order to always receive any new items. Secondly, the whole list can be retrieved using only one single short transaction; this is important, because the log is bound to change often, especially with method tracing. Having to keep a transaction open while the whole log is being read would lead to lots of concurrency conflicts. Thirdly, new items can easily be appended without having to traverse the entire list. For the *Traces* directory, we use a specialization of the general directory data structure shown earlier in this chapter.

The space log provides methods for prepending new log entries to the data structure, for getting the newest item (from which the previous items can be traversed), and for getting the full log, all in a thread-safe and transactional fashion. The data structure automatically ensures that the log items are deserialized before being returned to the user. Because they are not meant to be written more than once, it is not necessary to perform any further synchronization or to make any of their operations transactional; so for getting the full log, only the reading of the surrounding log structure must be transactional.

The actual data stored in the log data structure must be serializable for the space and is wrapped in a dedicated substructure of the log entry objects together with the date and time of creation for further reference.

```
public class Traces {
  private static readonly SpaceDirectory<LinearSpaceLog> Instance =
    SpaceDirectory<LinearSpaceLog>.GetOrCreate("Traces");
}

[WatchedSpaceObject]
[TransactionalObject]
public class LinearSpaceLog {
  private object monitor = new object(); // used for synchronizing multithreaded
    access
  private LogItem newestItem = null;

  [Transactional]
  public void AddData(CorsoShareable data) {
    lock (monitor) {
      newestItem = SpaceObjectFactory.CreateNewSpaceObject<LogItem>(new
        ItemData(data), newestItem);
    }
  }

  [Transactional(ShouldRefresh = true, ShouldPersist = false)]
  public LogItem GetNewestItem() {
    lock (monitor) {
      return newestItem;
    }
  }

  public IEnumerable<ItemData> GetFullLog() {
    List<ItemData> log = new List<ItemData>();
    LogItem item = GetNewestItem();
    while (item != null) {
      log.Insert(0, item.Data);
      item = item.Next;
    }
    return log;
  }
}

[SpaceObject]
public class LogItem {
  private ItemData data;
  private LogItem next;

  public LogItem(ItemData data, LogItem next) {
    this.data = data;
    this.next = next;
  }

  public ItemData Data {
    get { return data; }
  }

  public LogItem GetNext() {
    return next;
  }
}

[SpaceSerializable]
public class ItemData {
  public readonly DateTime CreationTime;
  public readonly CorsoShareable SharedData;

  public ItemData(CorsoShareable sharedData) {
    this.CreationTime = DateTime.Now;
    this.SharedData = sharedData;
  }
}
```

Listing 5.19: Space-based linear list data structure

With this data structure, the three tracing aspects are simple to implement. For each aspect, there is a serializable data class, which contains the trace

information to be stored in the log. The aspects themselves can be configured by ways of their factory attributes whether to be instantiated per class, per object, or as a singleton. On instantiation, each of them creates and registers a new log object, which is used for tracing, so the instantiation policy also influences whether there is one log per class, per object, or per application.

The actual tracing is performed within respective advice methods: instantiation tracing is performed in an *after construction* advice, error tracing in an *after exception* advice, and method executions are traced in an *around method* advice in order to be able to log both method entries and exits as well as return values. In each advice method, the tracing data (i.e. the method or type name, exception data, constructor arguments convertet to strings, etc.) is packaged into a data object and then added to the log structure. Code for the aspect implementations is given in listing 5.20.

The logs are never removed from the directory; they will persist until explicitly removed by an administrator or monitoring tool.

```
public enum TraceScope {
  PerObject,
  PerClass,
  PerApplication
}

[AttributeUsage(AttributeTarget.Class)]
public class TraceInstantiationsAttribute : Attribute {
  // holds one of the standard factory attributes for instantiation
  private AbstractSimpleFactoryAttribute factoryAttribute;

  public InstantiationTracedAttribute(TraceScope traceScope) {
    // prepare respective standard factory attribute for later aspect
      instantiation
    switch (traceScope) {
      case TraceScope.PerObject:
        factoryAttribute = new
          PerObjectAspectAttribute(typeof(InstantiationTracingAspect));
        break;
      case TraceScope.PerClass:
        factoryAttribute = new
          PerClassAspectAttribute(typeof(InstantiationTracingAspect));
        break;
      default:
        factoryAttribute = new
          SingletonAspectAttribute(typeof(InstantiationTracingAspect));
        break;
    }
  }

  [AspectFactory]
  public InstantiationTracingAspect GetInstance(Type targetType, object[]
    targetConstructorArgs) {
    return (InstantiationTracingAspect)factoryAttribute.GetInstance(targetType,
      targetConstructorArgs);
  }
}

[AttributeUsage(AttributeTarget.Class)]
public class TraceMethodsAttribute : Attribute { ... }

[AttributeUsage(AttributeTarget.Class)]
public class TraceErrorsAttribute : Attribute { ... }

public class InstantiationTracingAspect {
  [SpaceSerializable]
  public class TraceData {
    public string TypeName;
    public string[] ConstructorArguments;
  }
```

```
  private LinearSpaceLog log;

  public InstantiationTracingAspect() {
    log = SpaceObjectFactory.CreateNewSpaceObject<LinearSpaceLog>(new
      CorsoStrategy(CorsoStrategy.LAZY | CorsoStrategy.RELIABLE1));
    Traces.AddObject(log);
  }

  [Advice(typeof(AfterConstruction))]
  public void TraceConstruction(Type constructedType, object[] constructorArgs) {
    TraceData data = ObjectFactory.Create<TraceData>();
    data.TypeName = constructedType.FullName;
    data.ConstructorArguments = Array.Convert<object, string>(constructorArgs,
      delegate (object arg) { return arg.ToString(); });
    log.AddData((CorsoShareable)data);
  }
}

[AttributeUsage(AttributeTargets.Method)]
public class ExecutrionTracedAttribute : Attribute { }

public class MethodTracingAspect {
  [SpaceSerializable]
  public class EntryData {
    public int ThreadID;
    public string Target;
    public string MethodName;
    public string[] Arguments;
  }

  [SpaceSerializable]
  public class ExitData {
    public int ThreadID;
    public string MethodName;
    public string ReturnValue;
  }

  private LinearSpaceLog log;

  public MethodTracingAspect() {
    ...
  }

  [Advice(typeof(AroundMethod)), WhereDefined(typeof(ExecutionTracedAttribute))]
  public void TraceMethods(object target, MethodInfo method, object[] args,
    IMethodProceeder proceeder) {
    EntryData entry = ObjectFactory.Create<EntryData>();
    entry.ThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
    entry.Target = target.ToString();
    entry.MethodName = method.Name;
    entry.Arguments = Array.Convert<object, string>(args, delegate (object arg) {
      return arg.ToString(); });
    log.AddData((CorsoShareable)entry);

    object returnValue = null;
    try {
      returnValue = proceeder.Proceed(args);
      return returnValue;
    }
    finally {
      ExitData exit = ObjectFactory.Create<ExitData>();
      exit.ThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
      exit.MethodName = method.Name;
      if (returnValue == null) {
        exit.ReturnValue = "null";
      }
      else {
        exit.ReturnValue = returnValue.ToString();
      }
      log.AddData((CorsoShareable)exit);
    }
  }
}
```

```
[AttributeUsage(AttributeTargets.Method)]
public class ErrorTracedAttribute : Attribute { }

public class ErrorTracingAspect {
  [SpaceSerializable]
  public class TraceData {
    public int ThreadID;
    public string Target;
    public string MethodName;
    public string[] Arguments;
    public string Exception;
  }

  private LinearSpaceLog log;

  public ErrorTracingAspect() {
    ...
  }

  [Advice(typeof(AfterException)), WhereDefined(typeof(ErrorTracedAttribute))]
  public void TraceErrors(object target, MethodBase methodOrConstructor, object[]
    args, Exception exception) {
    TraceData data = ObjectFactory.Create<TraceData>();
    data.ThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
    if (target != null) {
      data.Target = target.ToString();
    }
    else {
      data.Target = "null";
    }
    data.MethodName = methodOrConstructor.Name;
    data.Arguments = Array.Convert<object, string>(args, delegate (object arg) {
      return arg.ToString(); });
    data.Exception = exception.ToString();
    log.AddData((CorsoShareable)data);

    throw exception;
  }
}
```

Listing 5.20: Tracing aspects and data structures

Example Usage Listing 5.21 shows how to use the tracing aspects on a simple business class. As the attribute declarations clearly communicate, the aspects are used to trace instantiations of *BusinessClass* in an application-wide log. Calls to and exceptions occurring in its *Operation1* method as well as errors occurring in its *Operation2* method are appended to a class-level log, and *Operation3* is not traced at all.

Evaluation

Code locality (CL): Code locality and separation of concerns as implemented by this aspect is excellent: all concern code is cleanly encapsulated into aspects and data structure classes. Every entity of modularization deals with exactly one concern.

Understandability (U): The fact that the factory attributes as well as the configuration attributes can serve as documentation increases understandability and readability of the code. By looking at the attribute declara-

```
[TraceInstantiations(TraceScope.PerApplication)]
[TraceMethods(TraceScope.PerClass)]
[TraceErrors(TraceScope.PerClass)]
public class BusinessClass {
  [Traced, ErrorTraced]
  public virtual void Operation1() {
    ...
  }

  [ErrorTraced]
  public virtual void Operation2() {
    ...
  }

  // not traced
  public virtual void Operation3() {
    ...
  }
}
```

Listing 5.21: Applying the tracing concern to a business class using aspects

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	4	3	4	5	5	5	4.5 (good)

Table 5.6: Evaluation matrix for tracing aspects

tions, it is immediately clear whether a class contains tracing code, what exactly is traced, and what is the scope of the log file.

Effective code size and effort (CSE): Effort for applying tracing to a class is very low, only one line of code is needed. For tracing of method executions and errors, an additional line of code is needed per traced method. While effort could be reduced by making the aspects log every call/error by default, this would yield very large and complex logs, so this way of explicit configuration was chosen instead.

Performance implications (PI): Some performance implications are to be expected for objects and methods whose instantiations respectively executions are traced, because data needs to be serialized and written to the space.

Changeability of target code (CT): Changes to the target code do not affect the concern, separate recompilation is possible. Care needs to be taken to update the configuration attributes when methods are added or changed.

Changeability of concern code (CC): Concern code can easily be changed without affecting target code, separate recompilation is possible.

Reusability of concern code (RC): The aspects are implemented in a way general enough for reusing them on any target class.

Transparency of composition (TC): The aspects have no side-effects or dependencies on other concerns.

Relevance for Space-Based Computing As indicated before, tracing generally is an important concern for diagnostic purposes. Tracing via the space offers additional advantages, this is not restricted to a specific middleware. The data structure details vary, of course; XVSM, for example, readily provides a FIFO container type, which is ideally suited for a linear log.

Heartbeat and Heartbeat-If

Goal and Usage Scenario One common requirement for distributed diagnostics of long-running applications and services is to check whether the respective processes are still on-line, and, if yes, whether it is still performing its job (as opposed to, for example, being stuck in a deadlock). A good solution for this issue is to have a *heartbeat*, i.e. a signal published at regular intervals, and a *heartbeat-if*, i.e. a signal published at regular intervals only if a certain condition is met. Such a heartbeat can also be associated with a specific object instance rather than a whole application to indicate that the object has not yet been garbage-collected.

If the heartbeat concerns (including the heartbeat-if variation) are implemented with the space-based computing paradigm, it requires a published space object which is regularly written by a monitored service or application. Monitoring software connected to the space can subscribe to the object's notification and watch the heartbeat similar to how a doctor can monitor a patient's pulse to check whether the person is alive. For best diagnostics, the data written to the heartbeat object should include the time at which the heartbeat was generated, so that an administrator can quickly find out for how long a process is dead (an improvement over real heartbeats).

Apart from the diagnostic use case, realizations of the heartbeat-if concern can also be used for *triggering* purposes: a process listening for the heartbeat can be triggered to take some action when the heartbeat stops (or starts), thus allowing for a means for event-based programming over the network.

Aspect-Oriented Realization

Design To realize the heartbeat concerns in an aspect-oriented way, we will use an aspect which incorporates an object with space identity and a background thread regularly persisting this object. Being applied to a business class in a per-object fashion, the background thread is established at object construction time and halted only when the object (and thus the aspect) is finalized. The object being persisted is a very simple class structure holding only the aforementioned timestamp of the heartbeat.

To include the heartbeat-if concern into the aspect, we define an additional *HeartbeatConditionAttribute*, which can be applied to boolean fields and properties. The aspect will check all suchly annotated members and will issue the heartbeat only if all yield *true*.

Publication of the heartbeat objects will be done via a *Heartbeats* specialization of the directory structure shown earlier in this chapter.

Implementation For the implementation, we first define a heartbeat class as illustrated in listing 5.22. The heartbeat class constitutes the object being written into the space and also incorporates the pulse thread, which periodically persists the object. Upon instantiation, objects of the class are passed the pulse period (i.e. the time that should pass between two subsequent heartbeats) as well as a set of condition fields and properties. The constructor immediately sets up and starts the background thread, which is only stopped (i.e. interrupted and then joined) when the object is disposed or when the application finishes.

The thread simply consists of a flag-terminated loop, which alternately triggers the pulse and sleeps for the specified pulse period. The pulse only results in the object to be updated and persisted if all condition members yield *true*.

The aspect simply instantiates a heartbeat object in its constructor, registering it in the *Heartbeats* directory, and disposes it in its finalizer, which is called when the aspect and target object are finalized prior to being garbage collected. This is similar to the aspects already shown in this chapter and is not separately re-demonstrated here. The collection of condition fields is retrieved via XL-AOF's field injection feature, the properties are retrieved via the *AspectEnvironment*.

For readability and declarative configuration, the aspect again comes with a dedicated factory attribute, which instantiates the aspect in a per-object fashion (needed in order to correctly stop the heartbeat when the target object is claimed by the garbage collector) and which takes the heartbeat's pulse period as a declarative parameter.

Example Usage Usage of the aspect is demonstrated in listing 5.23, where a heartbeat is established for a typical main form class of a .NET *Windows Forms* application. When the aspect is applied to such a class, whose objects live throughout the application runtime, the heartbeat can be seen as a life signal for the whole process rather than single objects. The heartbeat is configured to come at a rate of once per two seconds, and it will only be issued if the *Online* property is set to true, which indicates that the application processed some data in the last five seconds. The heartbeat will therefore cease to be signaled if the application stops processing data, which could cause a system monitoring tool to issue an administrator warning or switch to a backup system.

Evaluation

Code locality (CL): Code locality of the heartbeat concern implementation is very good, we managed to implement both heartbeat and heartbeat-if (because the latter is a generalization of the first) in a cleanly modularized aspect.

Understandability (U): Due to the documentary nature of the aspect's factory and configuration attributes, it is easily understandable from the resulting code whether a class has a heartbeat or not, at what rate it comes, and what fields and properties are conditions for the pulse.

Effective code size and effort (CSE): Because the aspect cleanly encapsulates the aspect, code size and effort needed for adding the concern to a

```
[SpaceObject]
public class Heartbeat {
  private DateTime timeOfPulse = DateTime.MinValue;
  private TimeSpan pulsePeriod;
  [NotSpaceSerialized] private ReflectedFieldCollection<bool> conditionFields;
  [NotSpaceSerialized] private ReflectedPropertyCollection<bool>
    conditionProperties;
  [NotSpaceSerialized] private volatile bool shouldStop = false;
  [NotSpaceSerialized] private Thread heartbeatThread;

  public Heartbeat() : this(TimeSpan.FromSeconds(5.0), null, null) {
  }

  public Heartbeat(TimeSpan pulsePeriod, ReflectedFieldCollection<bool>
    conditionFields, ReflectedPropertyCollection<bool> conditionProperties) {
    this.conditionFields = conditionFields;
    this.conditionProperties = conditionProperties;
    this.pulsePeriod = pulsePeriod;
    heartbeatThread = new Thread(HeartbeatThread);
    heartbeatThread.IsBackground = true;
    heartbeatThread.Start();
  }

  public void Dispose() {
    shouldStop = true;
    heartbeatThread.Interrupt();
    heartbeatThread.Join();
  }

  public DateTime TimeOfLastPulse {
    get { return timeOfPulse; }
  }

  private void Pulse() {
    timeOfPulse = DateTime.Now();
    ((ISpaceObject)this).Persist(ConnectionManager.NetworkTimeout);
  }

  private void PulseIfConditions() {
    foreach (ReflectedField<bool> cond in conditionFields) {
      if (!cond.Value) {
        return;
      }
    }
    foreach (ReflectedProperty<bool> cond in conditionFields) {
      if (!cond.InvokeGetMethod()) {
        return;
      }
    }
    Pulse();
  }

  private void HeartbeatThread() {
    while (!shouldStop) {
      try {
        PulseIfConditions();
        Thread.Sleep(pulsePeriod);
      }
      catch (ThreadInterruptedException) {
        // ignore, interruption is only to stop sleeping
      }
    }
  }
}
```

Listing 5.22: *Heartbeat* class implementing the pulse thread

target class is very low. Just one line of code is needed per monitored class, and one additional line of code is needed per condition.

```
[HasHeartbeat(TimeSpan.FromSeconds(2.0).TotalMilliseconds)]
public class MainForm : Form {
  ...

  [HeartbeatCondition]
  public bool Online {
    get { return DateTime.Now - lastProcessedData.DateTime <=
      TimeSpan.Seconds(5.0); }
  }
}
```

Listing 5.23: Usage sample for the heartbeat aspect

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	4	5	5	5	4.875 (good)

Table 5.7: Evaluation matrix for heartbeat aspect

Performance implications (PI): Because the heartbeat is performed in the background, there are no adverse performance implications to be expected by the implementation of this concern.

Changeability of target code (CT): The target code can be changed without affecting the concern code, caution is only needed when a condition field or property changes. Separate compilation is possible.

Changeability of concern code (CC): The concern code can be changed without affecting the target code, separate compilation is possible.

Reusability of concern code (RC): The aspect is implemented in such a general way that it is applicable to any target class.

Transparency of composition (TC): The concern has no dependencies or negative implications on other concerns.

Relevance for Space-Based Computing The concern realization given in this section is not specifically designed for CORSO and is equally applicable to other space-based abstraction layers. However, the coordination data structures provided by advanced and modern technologies—such as XVSM's FIFO structure—may provide even better structural possibilities for the heartbeat object than the shared CORSO object we used as a realization.

5.3.2 Caching

Besides diagnostic and monitoring purposes, the space can also be used as a distributed data cache, temporarily or permanently holding data which is complex to calculate or to retrieve. If a space-based implementation supports intelligent replication, as CORSO and even more so XVSM do, cached data can be retrieved from the space much faster than it would take to reload or recalculate it. In this subsection, we will therefore describe two caching aspects, of which the first uses local memory and only illustrates the concept, whereas the second implements the same cache in the space. Further extensions of these concern (not shown here) could include a timeout-based cache content lifetime model.

Local Memory Cache

Goal and Usage Scenario Consider a method which performs a very long operation, e.g. calculating prime numbers, extracting a complicated query from a database, or retrieving an XML file via a slow web service. If this method is called multiple times with the same arguments, it will always return the same result, and it will always take the same long time. If the method is used from different places and at different times during program execution, it would therefore be sensible to cache its results based on method parameter indexing: store them in an in-memory table based on the parameter values. If used intelligently, this could significantly improve program performance without complicating the program source code.

Aspect-Oriented Realization

Design The aspect design for implementing this concern is simple: an *around method* advice is used to intercept all methods tagged to be cached with a dedicated declarative *CachedResultAttribute*. It passes the method arguments to a cache data structure (explained later), which performs the actual caching, and only executes the original method if the result has not yet been stored. A dedicated factory attribute instantiates the caching aspect on a singleton basis, resulting in a cache for the whole application. The class structure used for this aspect is shown in figure 5.7.

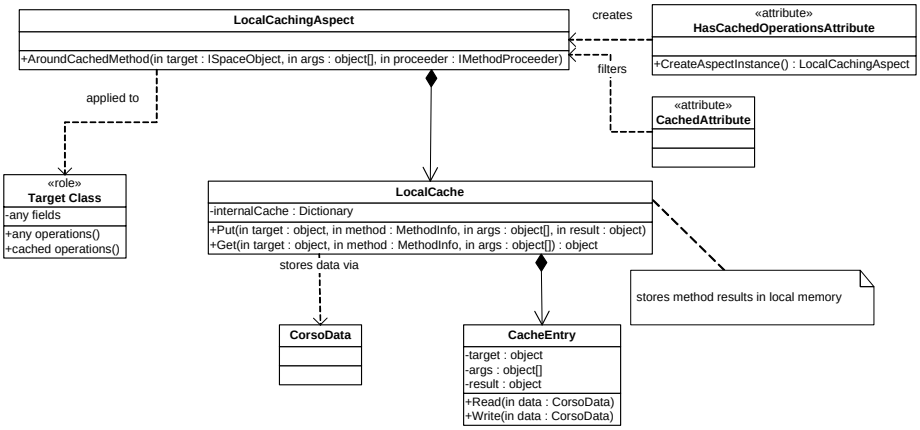

Figure 5.7: Class structure for a local caching aspect

Implementation Implementation-wise, the aspect is almost trivial—it only contains a single around advice bound to methods attributed with the tag attribute, which checks whether the cache data structure already holds the result for the given arguments, method, and target object. If yes, the result is returned without invoking the actual method; if no, the result is calculated by the method and subsequently stored in the cache, as illustrated in listing 5.24.

```
[AttributeUsage(AttributeTargets.Method)]
public class CachedAttribute : Attribute { }

public class LocalCachingAspect {
  private LocalCache cache = new LocalCache();

  [Advice(typeof(AroundMethod)), WhereDefined(CachedResultAttribute)]
  public object AroundCachedMethod(object target, Method method, object[] args,
    IMethodProceeder proceeder) {
    object result = cache.Get(target, method, args);
    if (result == null) {
      result = proceeder(args);
      cache.Put(target, method, args, result);
    }
    return result;
  }
}
```

Listing 5.24: Local memory caching aspect

The more complex issue is that of the cache structure, which contains results indexed by a target/method/arguments triple. These (as well as the results) cannot be stored by reference, but must be kept in an immutable representation in order to guarantee correct application behavior.

We will use the space serialization mechanism presented earlier in this chapter in order to serialize the target object, arguments, and results into a *CorsoData* object[2]. The serialized data will be stored in a internal multi-map [Win05] cache structure whose key is the full method name paired with the XOR-combined hash codes of target and argument objects. For a cache lookup, the respective CorsoData objects are retrieved and targets, arguments, and results are deserialized. If real target and arguments match the data of a deserialized object, the respective result is returned. The data structure's code is given in listing 5.25.

```
public class LocalCache {
  private IDictionary<Pair<string,int>,List<CorsoData>> internalCache = new
    Dictionary<Pair<string,int>,List<CorsoData>>();

  public void Put(object target, Method method, object[] args, object result) {
    CorsoData data = GetSerializedEntry(target, args, result);

    string fullName = GetFullName(method);
    int hashCode = GetCombinedHashCode(target, args);
    Pair<string,int> id = new Pair(fullName, hashCode);
    if (!internalCache.ContainsKey(id)) {
      internalCache[id] = new List<CorsoData>();
    }
    internalCache[id].Add(data);
  }

  public object Get(object target, Method method, object[] args) {
    string fullName = GetFullName(method);
    int hashCode = GetCombinedHashCode(target, args);
    Pair<string,int> id = new Pair(fullName, hashCode);
    if (internalCache.ContainsKey(id)) {
      CacheEntry entry = ObjectFactory.Create<CacheEntry>();
      foreach (CorsoData data in internalCache[id]) {
        ... // first reset the CorsoData object to its beginning; not shown here
        ((CorsoShareable)entry).Read(data);
```

[2]Usually, one would rather use the built-in .NET serialization mechanism, which is faster and doesn't depend on the CORSO middleware. CORSO-based serialization is used here in order to prepare the structure for use in a space-based cache, see the next concern.

```
        if (entry.Matches(target, args)) {
          return entry.Result;
        }
      }
      return null;
    }
    else {
      return null;
    }
  }

  private CorsoData GetSerializedEntry(object target, object[] args, object
    result) {
    CacheEntry entry = ObjectFactory.Create<CacheEntry>();
    entry.Target = target;
    entry.Args = args;
    entry.Result = result;
    CorsoData data = ConnectionManager.GetConnection().CreateData();
    ((CorsoShareable)entry).Write(data);
    return data;
  }

  private string GetFullName(MethodInfo method) {
    return method.DeclaringType.FullName + "." + method.Name;
  }

  private int GetCombinedHashCode(object target, object[] args) {
    int hashCode = target.GetHashCode();
    foreach (object arg in args) {
      hashCode ^= arg.GetHashCode();
    }
    return hashCode;
  }
}

[SpaceSerializable]
class CacheEntry {
  public object Target;
  public object[] Args;
  public object Result;

  public bool Matches(object target, object[] args) {
    if (!object.Equals(target, Target) || Args.Length != args.Length) {
      return false;
    }
    for (int i = 0; i < args.Length; ++i) {
      if (!object.Equals(args[i], Args[i]) {
        return false;
      }
    }
    return true;
  }
}
```

Listing 5.25: Cache structure for operation results

Example Usage Usage of the local memory caching aspect is demonstrated in listing 5.26, where the results of a supposedly long-running operation, which retrieves a large set of data from a database, are cached.

```
[HasCachedOperations]
public class BusinessClass {
  [Cached]
  public string[] GetDataFromDatabase(string id) {
    ... // execute a long-running query based on id and return the results
  }
}
```

Listing 5.26: Example usage of the local memory caching aspect

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	4	5	5	5	4.875 (good)

Table 5.8: Evaluation matrix for local caching aspect

Evaluation

Code locality (CL): The aspect-oriented realization of local memory caching makes for clean modularization and good separation of concerns, having the concerns cleanly encapsulated in their dedicated aspects and classes. There is one class holding all data structure code, an aspect holding all the join point-related code, and business code is not influenced by the concern code.

Understandability (U): The implementation with declarative attributes to indicate whether a class uses operation caching and the results of what methods are cached make for good understandability and self-documenting code.

Effective code size and effort (CSE): Code size and effort needed for applying the caching aspect to a business class are very low: one attribute to be applied to the class definition, plus one attribute per cached method.

Performance implications (PI): There are no adverse performance implications to be expected by this aspect; in fact, performance is bound to improve unless the aspect is wrongly applied to short-running methods. (In such wrong applications, performance degradation due to the additional cache management effort is to be expected.)

Changeability of target code (CT): Target code can be changed without affecting concern code, separate compilation is possible. Some caution needs to be taken when changing methods with cached results.

Changeability of concern code (CC): Concern code can change without the target code needing to change, separate compilation is possible.

Reusability of concern code (RC): The concern is implemented in such a general way that it can be applied to any target class.

Transparency of composition (TC): The concern does not depend on or affect any other aspect.

Distributed Results Cache

Goal and Usage Scenario As already indicated, extending the previous concern by using the space as a distributed results cache has the advantage that once the result has been calculated by one process connected to the space, all other processes can use it as well.

Aspect-Oriented Realization

Design Design-wise, the distributed result caching aspect is modeled completely the same way as its local counterpart presented in the previous section. The only difference is that the cached data is not stored in a local dictionary any longer, but instead in a distributed hash map.

With CORSO, there are different ways to implement such a data structure; the easiest (but not the most powerful one) being to employ CORSO's named object feature, using the cache's index data as name and storing the rest in a named object. More power and flexibility yields implementation of a real map coordination data structure.

Implementation The aspect part of the distributed cache is equivalent to the implementation of the local cache presented in the previous section. The critical point is implementation of the map data structure. Efficient space-based hash structures are not the topic of this work and are thus not discussed here, and powerful collection space-based collection utilities have already been implemented by others [Rie04]. However, we present an implementation of a CORSO-based *SpaceMap* data structure created in the course of our work in appendix B, which can be used to implement the internal cache dictionary in a space-based manner—in fact, it even implements the *IDictionary* interface used for the local cache in the previous section, so the necessary changes to the cache structure's code are minimal. For further discussion of distributed and efficient hashtables, we refer to—for example—"Kademlia: A Peer-to-peer Information System Based on the XOR Metric" by Petar Maymounkov and David Mazières [MM02].

Usage and Evaluation Usage of the space-based caching aspect is equivalent to the local caching aspect; therefore, we will not give a dedicated usage sample for it. Similarly, an additional evaluation wouldn't make sense, since the aspects' implementations are exactly the same (safe the internal data structure, which wouldn't be considered in the evaluation anyway). We therefore refer to the previous section for both usage example and concern evaluation.

Relevance for Space-Based Computing Caching concerns are important usage scenarios for space-based computing in general, not only for the CORSO middleware. However, whereas CORSO does not include a pre-built dictionary data structure for implementing this concern, it's successor XVSM provides such a data container out of the box, making implementation of the concern's backing data structure a breeze.

5.3.3 Error Handling

Goal and Usage Scenario An important issue that shouldn't be overlooked when implementing a distributed concern library for space-based computing is the group of general error handling concerns, and especially the dealing with

distribution-induced errors. The aspects presented so far abstract many distributional issues into reusable components, which present themselves to the user of the library only in the form of declarative annotations. However, behind the scenes, all of these concerns involve communication, possibly over unreliable networks. Communication with the local CORSO kernel is often reliable—in most scenarios, a kernel will be installed on the same machine on which the process is executed. Communication which involves inter-kernel communication (e.g. for primary copy migration), on the other hand, are potentially unsafe, and errors must be expected.

With the subset of CORSO's .NET &Co we use in our work, errors come in different flavors:

- *CorsoTimeoutExceptions* in most cases indicate that a remote kernel could not be reached in the given timeout period. In the aspects presented so far, global timeout values configured with the connection manager are used. If these are chosen sensibly, one can assume a network problem if such an exception arises. In some cases, this exception can also come in a controlled scenario; e.g. with blocking notifications or blocking read operations. The aspects always know how to handle this kind of exception in such cases, but they cannot handle general timeouts caused by network problems.

- *CorsoDataExceptions* indicate a problem with the data in the space. Usually, these are caused by user errors, e.g. deserialization code not matching the serialization code or a wrong type being assumed for deserialization of a space object. Generally, the aspects presented so far cannot handle these errors in a reusable fashion. In most cases, these errors point to a faulty process being in the system.

- *CorsoReadExceptions* and *CorsoWriteExceptions* occur when reading or writing of an object yields an error not caused by a network timeout. The aspects presented so far can't handle these errors either.

- *CorsoTransactionExceptions* indicate that a transaction cannot be committed due to concurrency issues. The transaction aspect introduced earlier in this chapter handles this error by repeating the transactional operation for a number of times, but in case of starvation, when the process never gets the chance of committing the transaction, it also gives up and propagates the error to the caller.

- *CorsoException,* finally, is not only the base class of all other exception classes, but is also directly instantiated for all other error situations. Reusable concern implementations cannot generally know how to handle these.

Therefore, an application using one or several of the aspects in this library will sooner or later come into the situation of having CORSO-related exceptions being thrown. Usually, error handling is application-dependent, so users of the library will have to write their own *try/catch* handlers or error handling aspects (which XL-AOF makes very easy due to its *after exception* join points).

However, there are three important error handling strategies, which are often used (sometimes even in combination) and which are preimplemented in this library:

- Logging the error to the console or a log file,
- silently swallowing the error, and/or
- displaying a user interface, informing the user that an error occured.

In fact, these strategies aren't only used for distribution-related errors, but for general error conditions, so reusable implementations of these strategies should be suitable for any exception kinds, not only for CORSO exceptions.

Aspect-Oriented Realization

Design For each of the three strategies, we provide one aspect:

- One aspect, which has an *after exception* advice for tracing the error—we use the in-built *System.Diagnostics.Trace* class for tracing—, and then rethrowing it,
- one aspect for swallowing the error, and
- one aspect for displaying a user interface.

Each of the aspects comes with a dedicated factory attribute for applying the aspect to target classes with singleton scope—the aspects do not store any state, so this is the most efficient way of instantiating them. In addition, the aspects provide tag attributes to be applied to those methods for which the aspects should handle escaping exceptions. Table 5.9 gives a summary of the aspect and attribute types for the error handling concerns.

Aspect	Factory Attribute	Method Tag
ErrorTracingAspect	TracesErrorsAttribute	TracedErrorsAttribute
SwallowErrorHandlingAspect	SwallowsErrorsAttribute	SwallowedErrorsAttribute
UIErrorHandlingAspect	HasErrorUIAttribute	ErrorUIAttribute

Table 5.9: Aspect and attribute classes for the error handling concerns

Implementation Putting the aspect design into implementation is straightforward—listing 5.27 shows the error tracing aspect's source code as an example, the other aspects are implemented in exactly the same way.

First, the code defines the singleton-scoped factory attribute, which always returns the same aspect instance when XL-AOF calls its *GetInstance* method. The method tagging attribute defined next provides a way of specifying the exception type the aspect should be able to handle. The tag can also be applied to the same method multiple times—that way, the aspect can react on multiple exception types on one method.

Then, the aspect itself is defined: it only consists of one *after exception* advice. The advice cannot use the *WhereException* filter provided by XL-AOF, but instead has to manually extract the configuration attribute from the method object and check whether the thrown exception matches the configured exception types. This workaround is made necessary due to a limitation of the .NET platform at the time of writing: attributes cannot reference type variables. (Without this limitation, the aspect could be made a generic type, making the manual check unnecessary.) If the exception matches one of the configured types, a message is written to the trace stream; in any case, the exception is rethrown.

The aspect definition includes a precedence declaration over the error swallowing aspect. This causes errors to always be logged even if they are swallowed by the other aspect. The swallowing aspect, on the other hand, has precedence over the user interface aspect: this causes the UI not to be shown if the error is swallowed.

```
[AttributeUsage(AttributeTargets.Class)]
public class TracesErrorsAttribute : Attribute {
  private static ErrorTracingAspect aspect =
    ObjectFactory.Create<ErrorTracingAspect>();

  [AspectFactory]
  public ErrorTracingAspect GetInstance() { return aspect; }
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class TracedErrorsAttribute : Attribute {
  private Type exceptionType;

  public TracedErrorsAttribute(Type exceptionType) {
    this.exceptionType = exceptionType;
  }

  public Type ExceptionType {
    get { return exceptionType; }
  }
}

[Precedence(typeof(SwallowErrorHandlingAspect))]
public class ErrorTracingAspect {
  [Advice(typeof(AfterException))]
  [WhereDefined(TracedErrorsAttribute)]
  public object PerformTraceAfterException(MethodInfo method, Exception
    exception) {
    foreach (TracedErrorsAttribute attribute in
      method.GetCustomAttributes(typeof(TracedErrorsAttribute))) {
      if (attribute.ExceptionType.IsAssignableFrom(exception.GetType())) {
        System.Diagnostic.Trace.WriteLine("An error of type " +
          exception.GetType() + " occurred while executing the method " +
          method.DeclaringType.FullName + "." + method.Name + ": " +
          exception.Message + ".", "ERR");
      }
    }
    throw exception;
  }
}
```

Listing 5.27: UI error handling aspect

The DCL implementation also contains a second set of these aspects which are not configurable per method, but per class. Instead of handling only exceptions thrown from tagged methods, they handle exceptions thrown from any method of the target class. Their design and implementation equals that of the

method-level aspects (excluding the tag attribute definition, of course), only that the advice is not filtered at all. Instead, the exception type configuration is performed via the factory attribute.

Example Usage Listing 5.28 shows an example business class, whose first method logs and then swallows all CORSO-induced errors. The class also shows a user interface for all *InvalidOperationExceptions* thrown from the class.

```
[HasClassErrorUI(typeof(InvalidOperationException))]
[TracesErrors, SwallowsErrors]
public class BusinessClass {
  [TracedErrors(typeof(CorsoException)), SwallowedErrors(typeof(CorsoException))]
  public virtual void Operation1() {
    ...
  }
}
```

Listing 5.28: Usage example for the error handling aspects

Evaluation

Code locality (CL): These concerns show how nicely error handling code can be modularized—all error handling code is cleanly encapsulated into dedicated aspects.

Understandability (U): Because of the documentary effect of the well-named factory and configuration attributes, understandability of which classes have error handling and what exceptions are handled in which way is very good.

Effective code size and effort (CSE): Code size and effort needed to apply these aspects to a target class is very low. The method-level aspects require one attribute per class and one attribute for those methods requiring error handling. The class-level aspects require only an attribute per class.

Performance implications (PI): No adverse performance implications are to be expected by these concerns and their implementations.

Changeability of target code (CT): The target objects' method bodies can be changed independently from aspect code, and separate recompilation is possible. Nonetheless, error handling aspects are conceptually coupled to their target methods: it is necessary to consider what exceptions could be thrown from the target's methods and adapt the aspect applications accordingly whenever a method is changed.

Changeability of concern code (CC): The concerns' implementations can change without affecting the target code.

Reusability of concern code (RC): The concerns are implemented in such a general way that they can easily be applied to any target class.

Transparency of composition (TC): Error handling aspects automatically have an effect on other error handling aspects. The aspects included in the distributed concern library have a predefined precedence setting to

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	4	5	5	4	4.75 (good)

Table 5.10: Evaluation matrix for error handling aspects

correctly resolve this, but when they are to be composed with other error handling concerns, this must be considered. They have no effects on other concerns.

Relevance for Space-Based Computing The aspects as implemented in this section are not specifically targeted at CORSO. In fact, they can be used for any kind of exceptions occurring during program execution. Unfortunately, error conditions automatically come with distributed applications, so these concerns arise with any (distributed) space implementation.

5.3.4 Extensions of Low-Level Concerns

In this section, we will present higher-level extensions of the low-level concerns presented in section 5.2. The extension aspects augment the base aspects of notification handling, serialization, etc., adding additional features and useful functionality. In other words, the aspects presented in this chapter do not provide readily applicable solutions for high-level use cases (as the monitoring and caching aspects did), but instead provide building blocks to facilitate implementation of space-based applications.

Auto-Refreshed Object *(extends Notification)*

Goal and Usage Scenario The notification aspect adds an interface to a target object, enabling the user of the object to subscribe to asynchronous change notifications from the space. When fired, the asynchronous notification event provides a *CorsoData* object containing the new data of the modified object.

Often, it is desirable for the .NET representation of a CORSO object to always reflect the most recent space data in near real-time without need of transactional refresh operations—the object's internal state should always automatically be updated whenever the space data changes. This is easily solvable by using the notification aspect and subscribing to its notification event, but since this *auto-refreshed object* concern is a separate, cross-cutting concern built on top of the notification aspect, it should be separately encapsulated.

Aspect-Oriented Realization

Design The concern realization consists of just one aspect, which—being instantiated once per object—subscribes to the event provided by the notification aspect (or any other implementation of *IWatchedSpaceObject*). When the event

is fired, the aspect employs the target object's *Read* method (supplied by the serialization aspect or any other implementation of the *CorsoShareable* interface) in order to read the data sent with the notification event into the .NET object. The aspect comes with a dedicated factory attribute implementing the correct instantiation scope, and it leaves the task of starting and stopping the notification processing to the user in order provide most flexibility.

The behavior of automatic refreshing occurs asynchronously, on a background thread. This is bound to cause problems in case the object is currently in use. In particular, problems would arise if an object were automatically refreshed while being engaged in a space transaction: due to the nature of networking, the notification data could actually be older than the data read with the transactional refresh operation. Therefore, two guarding mechanisms are required: a user must be able to lock an object, causing any asynchronous refreshes to be postponed until the lock is released, and the aspect must automatically postpone automatic refreshes for objects being used in a transaction.

For the former, we provide a custom attribute, which can be applied to a field of the target class, which constitutes the lock object. The aspect will synchronize on this object whenever an automatic refresh is to be made; if the user also locks this object to guard a code block, this will result in mutual exclusion of automatic refreshes with this code block.

For the latter, the aspect will use the extensibility join points provided by the transaction aspect. Before an object enters a transaction (and before the transaction aspect actively refreshes the object, if necessary), the automatic refresh aspect enlists the object for not being updateable. All automatic refresh notifications will be enqueued and postponed until the object leaves the transaction (and after the transaction aspect has persisted the object, if necessary).

Figure 5.8 shows the entities involved in the aspect-oriented design for the auto-refreshed object.

Implementation Implementation of the actual refreshing is straight-forward: after construction of the object, the aspect registers for the asynchronous change notification event. In the event handler, it first locks the target's lock field (if any available) and calls the object's *Read* method with the data delivered by the event. In fact, the aspect uses a reflected field collection rather than a single field for the lock, so the class supports multiple lock fields. Multiple locks should be used with care, however, because they can quickly lead to deadlocks if taken in inconsistent order.

For handling the mutual exclusion between transactional operations and automatic refreshes, we use a separate lock private to the aspect. The lock is acquired each time the target object enters a transaction and released each time it leaves a transaction. The notification event handler also acquires the lock and, since it is called on a background thread, will thus automatically wait until the object has left all transactions. Listing 5.29 contains the full aspect implementation (excluding the factory attribute, which is equivalent to those already presented).

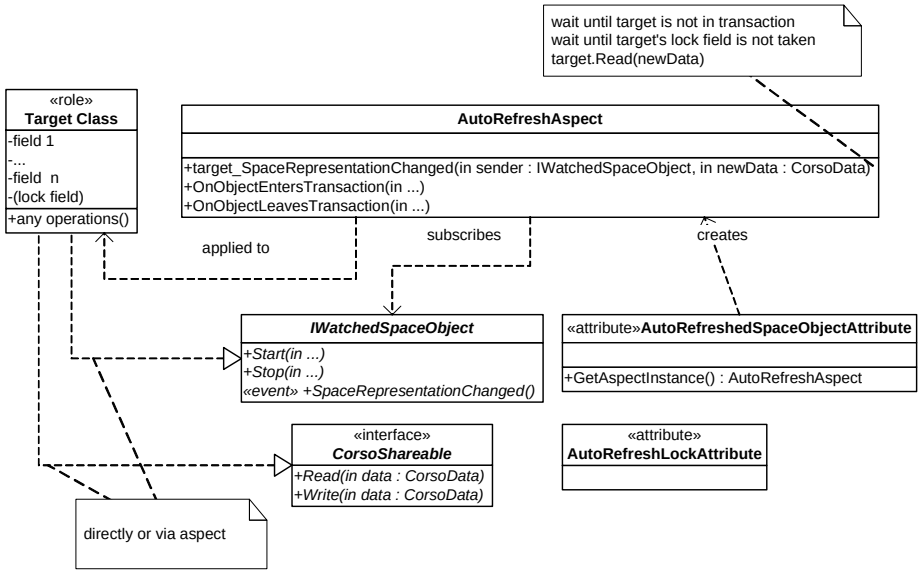

Figure 5.8: Aspect-oriented design of the auto-refreshed object concern

```
[AttributeUsage(AttributeTargets.Field)]
public class AutoRefreshLockAttribute : Attribute { }

[RequiresPerObjectAspect(typeof(WatchedSpaceObjectAspect))]
public class AutoRefreshAspect {
  [WhereDefined(typeof(AutoRefreshLockAttribute))]
  private ReflectedFieldsCollection<object> targetLocks =
    ReflectedFieldsCollection<object>.Injected;

  [Advice(typeof(AfterConstruction))]
  public void Initialize(object constructedInstance) {
    ((IWatchedSpaceObject)constructedInstance).SpaceRepresentationChanged += new
      SpaceRepresentationChangedHandler(target_SpaceRepresentationChanged);
  }

  private void target_SpaceRepresentationChanged(IWatchedSpaceObject sender,
    CorsoData newData) {
    AcquireLocks();
    ((CorsoShareable)sender).Read(newData);
    ReleaseLocks();
  }

  private object txLock = new object();

  private void AcquireLocks() {
    // always take transaction lock first, then specific locks
    Monitor.Enter(txLock);
    foreach (ReflectedField<object> lockField in targetLocks) {
      Monitor.Enter(lockField.Value);
    }
  }

  private void ReleaseLocks() {
    // as we release all locks in one go, order doesn't matter
    foreach (ReflectedField<object> lockField in targetLocks) {
      Monitor.Exit(lockField.Value);
    }
    Monitor.Exit(txLock);
  }
```

```
  [Advice(typeof(ObjectEntersTransaction))]
  public void OnObjectEntersTransaction() {
    Monitor.Enter(txLock);
  }

  [Advice(typeof(ObjectExitsTransaction))]
  public void OnObjectExitsTransaction() {
    Monitor.Exit(txLock);
  }
}
```

Listing 5.29: Implementation of the auto-refreshed object aspect

Example Usage Listing 5.30 shows an example applying the auto-refreshed object aspect to a business class. When the *IWatchedSpaceObject.Start* method is called on an instance of this class, the object starts to be synchronized with the space in near real-time. The business class guards its *DoubleState* property against concurrent automatic refreshes by using an attributed lock field. The transactional method *AppendState* method is automatically guarded against concurrent automatic refreshes. The automatic synchronization ends when the instance is explicitly removed from the notification manager using the *IWatchedSpaceObject.Stop* method.

```
[AutoRefreshedSpaceObject]
[TransactionalObject]
public class BusinessClass {
  private string state = "";
  [AutoRefreshLock] private object autoRefreshLock = new object();

  public string DoubleState {
    get {
      lock (autoRefreshLock) { return state + state; }
    }
  }

  [Transactional]
  public virtual void AppendState(string newState) {
    state += newState;
  }
}
```

Listing 5.30: Example usage of the auto-refreshed object aspect

Evaluation

Code locality (CL): The aspect manages to cleanly encapsulate all extensions of the asynchronous notification aspect that are required by the auto-refreshed object concern, concern code is nicely separated from the target code.

Understandability (U): Because of the documentary means of the dedicated factory attribute, it is very easy to understand whether an object is automatically refreshed or not. The *AutoRefreshLockAttribute* applied to fields used to guard against concurrent automatic refreshing also has a clear and easily understandable meaning.

Effective code size and effort (CSE): Only a single attribute (i.e. a single line of code) is necessary to make an object automatically refreshed. If

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	5	5	5	4	4.875 (good)

Table 5.11: Evaluation matrix for auto-refreshed object aspect

explicit guarding against concurrent refresh operations is needed, one additional line of code is required in order to tag the respective field.

Performance implications (PI): There are no adverse performance implications to be expected by this aspect.

Changeability of target code (CT): Target code is easily changeable without affecting the concern code, separate recompilation is possible.

Changeability of concern code (CC): Changes to the aspect code do not interfere with target code, separate recompilation is possible.

Reusability of concern code (RC): The aspect is implemented in such a general way that it is applicable to any target object with a space identity.

Transparency of composition (TC): The aspect requires an implementation of *IWatchedObject* and *CorsoShareable* to be present on its target classes and automatically adds the respective aspects to the class if necessary. In addition, it can cause the transaction aspect to block if an automatic refresh operation is currently conducted. It has, however, no adverse implications on other concerns.

Relevance for Space-Based Computing The auto-refreshed object concern is a good extension of the asynchonous change notification aspect for any space-based implementation. The caveats noted for the notification aspect in section 5.2.4 of course also apply to this extension.

Pooling *(extends OID Retrieval)*

Goal and Usage Scenario The space object factory implementing the OID retrieval, as described in section 5.2.3, does not implement any pooling. This means that every time it is asked to create an object for an existing OID or a named object, it will create a new local object reference. While this can be exactly the desired behavior, some use case scenarios would require the factory to keep track of the objects it created in an object pool and always return the same object instances for the same CORSO OIDs.

Aspect-Oriented Realization

Design An aspect implementing this concern is easily created by advising the object creation join point of classes for which pooling is desired. As indicated earlier in this chapter, the space object factory passes the OID of the object to be created as additional context information to this join point, so the aspect

can extract it and use it to look up objects from a pool of instances created earlier.

As with the other aspects, a specialized factory attribute (with singleton semantics) is provided for better readability.

Implementation The implementation given in listing 5.31 follows directly from the design. For the pooling, we use a dictionary object, mapping *CorsoVarOid* objects to object references. To avoid memory leaks, it is important to only use *weak references*; these do not keep the referenced objects in memory and are cleared when the objects are claimed by the garbage collector. In the listing, a *WeakDictionary* object is used, which automatically handles the management of weak references, scanning the dictionary for unused objects on every 50'th access.

When a pooled object is requested with a different type than previously used, an exception is thrown. If no OID context information is available (e.g. if another factory is used to create the object), the pool is bypassed.

```
public class PoolingAspect {
  private const int dictionaryScanThreshold = 50;
  private WeakDictionary<CorsoVarOid, object> pooledObjects = new
    WeakDictionary<CorsoVarOid, object>(dictionaryScanThreshold);

  [Advice(typeof(ObjectCreation))]
  public object ObjectCreated(Type t, object[] constructorArgs,
    IObjectInstanceCreator creator, IJoinpointInfo context) {
    if (context.AdditionalContext.ContainsKey("OID")) {
      object pooledObject = pooledObjects[context.AdditionalContext["OID"]];
      if (pooledObject == null) {
        pooledObject = creator.CreateInstance(t, constructorArgs);
        pooledObjects[context.AdditionalContext["OID"]] = pooledObject;
      }
      else if (!t.IsAssignableFrom(pooledObject.GetType())) {
        throw new InvalidOperationException("This OID was previously used as an
          object of type " + pooledObject.GetType().FullName +
          " and therefore can't be used for an object of type " + t.FullName +
          ".");
      }
      else {
        return pooledObject;
      }
    }
    else {
      return creator.CreateInstance(t, constructorArgs);
    }
  }
}
```

Listing 5.31: Implementation of the pooling aspect

Example Usage Usage of this aspect directly integrates into the object factory: by simply attributing a target class with the *PooledAttribute*, the space object factory will reuse instances with the same OIDs.

Evaluation

Code locality (CL): The complete pooling code is contained in a single aspect definition, which holds no other concern code, resulting in perfect separation of concerns and code locality.

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	5	5	5	4	4.875 (good)

Table 5.12: Evaluation matrix for pooling aspect

Understandability (U): Application of the dedicated factory attribute to a target class makes it easily understandable that it is influenced by the pooling concern.

Effective code size and effort (CSE): Adding pooling to a class only requires it to be attributed with one single attribute.

Performance implications (PI): There are no significant performance implications caused by this concern or its implementation.

Changeability of target code (CT): Target code can be changed without affecting the concern implementation and separate compilation is possible.

Changeability of concern code (CC): Concern code can be changed without affecting the target code and separate compilation is possible.

Reusability of concern code (RC): The concern is implemented in such a general way that it is readily applicable for any object with space-based identity.

Transparency of composition (TC): The concern has no direct effects on other concerns. Due to its pooling nature, it could however affect other object creation aspects, whose advice methods might not be called when the aspect decides to reuse an existing object.

Relevance for Space-Based Computing Whenever data objects with identity are used—no matter whether there is an implicit identity, like with CORSO objects or XVSM containers, or whether it must be explicitly modeled using primary keys, as with JavaSpaces—, tracking and pooling of object instances becomes of importance.

Lifetime Tracking *(extends Pooling)*

Goal and Usage Scenario One important issue in distributed systems is lifetime management of the data stored in the system. CORSO provides a garbage collection mechanism for unnamed space data objects: it counts the number of processes that directly or indirectly reference an object. When that sum reaches zero, the space data objects are removed. While this works well for applications with short-lived processes, it might introduce memory problems with long-lived processes, because CORSO cannot free any objects that were referenced by such a process process at any time until it gets shut down.

For this, CORSO supports manually signaling that a process doesn't need a space object any longer by invoking its *CorsoVarOid.Free* method. In .NET-based applications which use pooling and want to actively help in space memory management, that method should be called for a space data object as soon as the local object representing that space data object isn't used any longer.

Aspect-Oriented Realization

Design The .NET garbage collector has a kind of notification mechanism, notifying an object that it is about to be claimed as garbage: an object's *finalizer* can contain code that is executed at that time. If an object has an associated *per object scope* aspect and that aspect implements a finalizer, the aspect's finalizer will be called at the same time (or shortly before or after) the object's finalizer has run. We therefore use such an aspect as a lifetime tracker, which calls *Free* on its target object's OID when the target is about to be claimed as garbage. We also provide a special factory attribute, *LifetimeTracked*, which applies the aspect to a target object with per object scope.

Implementation Listing 5.32 shows the implementation of the lifetime tracking aspect. It is a straight-forward implementation of the design: the aspect holds an injected reference to the target object, by which it retrieves the target's OID and frees it when the finalizer is run.

```
public class LifetimeTrackerAspect {
  [Target]
  private ISpaceObject target = ReflectedField<object>.InjectedTarget;

  ~LifetimeTrackerAspect() {
    target.Oid.Free();
    target.Oid = null;
  }
}
```

Listing 5.32: Implementation of the lifetime tracking aspect

Example Usage Usage of this aspect directly integrates with the pooling aspect: by simply attributing a target object with the *LifetimeTrackedAttribute*, its life time will be tracked by the aspect and the OID will be freed on destruction. This aspect should only be used with pooled objects, because for others, there might be more than one object instance referencing the same space data object. Freeing the space object when one of these instances is garbage-collected could therefore be very wrong.

Evaluation

Code locality (CL): The whole lifetime tracking concern is cleanly modularized in a (very short) aspect, which contains no other concern code, yielding ideal separation of concern and code locality.

Understandability (U): By making use of the documentary features of the dedicated factory attribute, it is easily understandable that an object's lifetime is tracked by looking at its class definition.

Effective code size and effort (CSE): Tracking only requires one attribute to be added to a target object's class.

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	4	5	5	5	4	4.75 (good)

Table 5.13: Evaluation matrix for lifetime tracking aspect

Performance implications (PI): .NET objects with a finalizer are claimed in two steps (one for executing the finalizer, one for the actual claiming), which might slightly decrease performance in rare situations of very high memory pressure or in scenarios with dozens to hundreds of objects being created per second.

Changeability of target code (CT): Target code can be changed without any effect on the concern implementation.

Changeability of concern code (CC): Concern code can be changed without affecting target code.

Reusability of concern code (RC): The aspect is implemented to be applied to any pooled object with space identity.

Transparency of composition (TC): The implementation of this concern depends on object pooling; in cases without pooling, it might yield erroneous results. Apart from that, there are no dependencies or adverse effects on other aspects.

Relevance for Space-Based Computing While advanced space-based implementations such as XVSM offer different lifetime service concepts, such as lease times as well as reference tracking, explicit memory management is an issue also with these network abstractions.

Up-to-Dateness Service Level Agreement *(extends Shared Identity)*

Goal and Usage Scenario With the auto-refreshed object concern, we have described a solution for .NET objects being automatically refreshed whenever the space data associated with them changes. The transactional operations aspect also presented earlier in this chapter provides a mechanism to automatically call an object's refresh operation before a transactional operation is executed. Both of these concerns aim at keeping the .NET object always up-to-date with the space.

However, there are situations in which this isn't necessary, and even counterproductive: sometimes, an object need not be completely up-to-date, its data must only be *sufficiently recent*. In such cases, synchronizing an object with its space representation at every call of the *Refresh* operation (e.g. at the beginning of every transactional operation) would be a waste of bandwidth and should therefore be avoided. Instead, a user should be able to configure an up-to-datenesss service level agreement for classes with space-based identity, and refresh operations should simply be ignored if not really necessary.

Aspect-Oriented Realization

Design A realization of this concern must fulfill two requirements. First, it needs to provide a way of configuring what "sufficiently recent" means for a class with space-based identity. For this, it must allow the programmer to specify a time period defining how old an object's data may be to be acceptable. Second, it has to cause *Refresh* operations to be ignored if sufficiently recent data is already available.

The second requirement can be solved via an aspect with an *around method* advice bound to the *Refresh* methods of the target class. These are implemented on the class either directly or via the space identity aspect and can thus be intercepted by an around advice as any other method of the target object can. The advice can store the date and time of invocation in a member variable of the aspect and thus calculate the time passed since the last refresh operation. If the passed time is greater then the configured time period (or if the object is refreshed for the first time), it allows the *Refresh* method to be executed. Else, it simply returns without proceeding to the method.

The configuration requirement is solved by providing a custom factory attribute for the aspect: the attribute's constructor can be used to specify the time period, and the attribute can pass it on to the aspect when instantiating it. Since the aspect needs to store object-related information (the date and time of the last refresh), the attribute instantiates the aspect with per-object scope.

Implementation Implementation of this aspect is straight-forward, as shown in listing 5.33. The listing shows both the aspect's factory attribute, which allows configuration of the up-to-dateness time period and instantiates the aspect with per-object scope (passing the time period to the aspect's constructor), and the aspect itself. The aspect contains one around advice, which is bound to the *Refresh* methods declared by the *ISpaceObject* interface. The advice calculates the time passed since the last refresh operation and only proceeds if it exceeds the configured time period (or the object hasn't been refreshed yet). The aspect does not require (and silently add) any other aspects: if the *Refresh* methods of the *ISpaceObject* interface aren't present, it simply won't do anything.

Example Usage Listing 5.34 shows a business class whose data will only be refreshed every five minutes. This applies to manual calls to the *Refresh* method as well as to those automatically performed by the transactional operations aspect.

Evaluation

Code locality (CL): The aspect manages to cleanly encapsulate the code for the service level agreement concern and completely separate it from any other concerns.

Understandability (U): The meaning and configuration of this aspect as well as whether it is applied to a target class is easily understand by looking at the list of attributes applied to the target class.

```
[AttributeUsage(AttributeTarget.Class)]
public class UpToDatenessSLAAttribute : Attribute {
  private TimeSpan timePeriod;

  public UpToDatenessSLAAttribute(TimeSpan upToDatenessPeriod) {
    this.timePeriod = upToDatenessPeriod;
  }

  [AspectFactory]
  public UpToDatenessSLAAspect CreateInstance() {
    return ObjectFactory.Create<UpToDatenessSLAAspect>(timePeriod);
  }
}

public class UpToDatenessSLAAspect {
  private TimeSpan timePeriod;
  private DateTime lastRefresh = DateTime.MinValue;

  public UpToDatenessSLAAspect(TimeSpan timePeriod) {
    this.timePeriod = timePeriod;
  }

  [Advice(typeof(AroundMethod), WhereNameEquals("Refresh"),
    WhereDeclaringTypeEquals(typeof(ISpaceObject)))]
  public object AroundRefresh(IMethodProceeder proceeder, object[] args) {
    DateTime now = DateTime.Now;
    TimeSpan timePassed = now - lastRefresh;
    if (timePassed > timePeriod || lastRefresh == DateTime.MinValue) {
      lastRefresh = now;
      proceeder.Proceed(args);
    }
    return null; // Refresh methods are void and do not return a value
  }
}
```

Listing 5.33: Implementation of the service level agreement aspect

Effective code size and effort (CSE): Effort for applying this aspect to the target class comprises the addition of one single line of code: the attribute application to the target class.

Performance implications (PI): There are no adverse performance implications to be expected by this aspect. In fact, the concern itself is bound to improve application performance by reducing the amount of data sent on the network.

Changeability of target code (CT): The target code can be changed without affecting the concern, separate recompilation is possible.

Changeability of concern code (CC): Concern code can also be changed without affecting the target code, separate recompilation is possible.

Reusability of concern code (RC): The aspect is implemented in such a general way (being bound to the *Refresh* methods rather than use-case specific target class methods) that it can be applied to any target class.

Transparency of composition (TC): The aspect does not require any other aspects to be present on the target class (although it does nothing if there is no implementation of the *ISpaceObject* interface present) and has no effects on any other concerns apart from the space-based identity concern, for whom it loosens the refresh operation semantics (which is the point of the concern).

```
[SpaceObject]
[TransactionalObject, UpToDatenessSLA(TimeSpan.FromMinutes(5))]
public class BusinessClass {
  private string state;

  public string State {
    get { return state; }
  }

  [Transactional]
  public virtual void ChangeState(string additionalText) {
    state += additionalText;
  }
}
```

Listing 5.34: Example usage of the up-to-dateness service level agreement aspect

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	5	5	5	5	5 (excellent)

Table 5.14: Evaluation matrix for up-to-dateness service level agreement aspect

Relevance for Space-Based Computing Service-level agreements such as this one are an interesting feature for any distributed application, not only space-based ones, and for any space-based platform, not only CORSO. In fact, sophisticated space-based network abstractions such as *XVSM* can even provide service-level agreements on the middleware level, which facilitates development of higher level semantics otherwise hard to implement.

Compression/Encryption *(extends Object Serialization)*

Goal and Usage Scenario Space-based computing is based on the notion of a common virtual shared memory. In principle, every process participating in the space-based system can access all the data items it discovers in this virtual shared memory. Of course, access control and security are most important issues under these circumstances. While most space-based implementations provide some sort of access control (CORSO has support for pluggable encryption and access control modules, XVSM supports dynamically configurable, fine-grained security function profiles through its interception mechanism), security must sometimes be handled by the application itself. For such cases, an application can choose to manually encrypt its data just when it is serialized into the space, and decrypt it when it is deserialized.

Similarly, if an application needs to share large data items with other processes on the network, it might want to compress the serialized data, possibly saving a lot of bandwidth, and decompress it when it's being deserialized.

Although intended for different purposes, these two concerns both deal with manipulating the application's data at the boundaries to the space. We will therefore realize them together as if they were a single concern.

Aspect-Oriented Realization

Design The concerns can be realized as an aspect by extending the serialization aspect, reacting on the custom *before field serialization* and *before field deserialization* join points it exposes. The aspect is used to encapsulate the serializer automatically detected by the serialization aspect within another serializer implementation, which streams the data to an encryption or compression mechanism, performing the actual encryption/decryption or compression/decompression tasks.

Figure 5.9 shows the design of the concern realization. We provide a generic serialization extension aspect, which is bound to the join points exposed by the serialization aspect. It delegates to an implementation of the *ISerializationExtension* interface, which builds a serializer wrapper encapsulating the serializer chosen by the serialization infrastructure. There are two different implementations of the interface, one returns an encrypting wrapper, one returns a decrypting wrapper. The aspect is configured and instantiated by two different factory attributes: one configures the aspect to use the encryption extension, the other uses the compression extension. Each attribute instantiates the aspect as its own singleton, i.e. there is one single instance for the aspect configured for encryption and one single instance for the aspect configured for compression. Of course, it is possible to apply both attributes to a class, resulting two wrappers to be generated (one wrapping the other, which in turn wraps the original serializer), achieving encryption and compression of data at the same time.

Implementation Implementation of the aspect is straight-forward, listing 5.35 shows how the design presented in the previous section can be realized using XL-AOF. The listing shows only the aspect, as the factory attributes and the wrappers are trivial to implement and the two serialization extensions can be realized using readily available .NET framework and third-party encryption and compression components. The single advice method can be used for both serialization join points because they have the same signature.

```
public class SerializationExtensionAspect {
  private ISerializationExtension extension;

  public SerializationExtensionAspect(ISerializationExtension extension) {
    this.extension = extension;
  }

  [Advice(typeof(BeforeFieldSerialization))]
  [Advice(typeof(BeforeFieldDeserialization))]
  public void BeforeFieldSerialization(ref ISpaceSerializer serializer) {
    // change the serializer to be a wrapper
    serializer = extension.BuildWrapper(serializer);
  }
  }
```

Listing 5.35: Compression and encryption of space data realized as an aspect

Example Usage Listing 5.36 shows an example application of both the encrypting and the compressing variant of the aspect to a business class which contains highly redundant and very sensitive data.

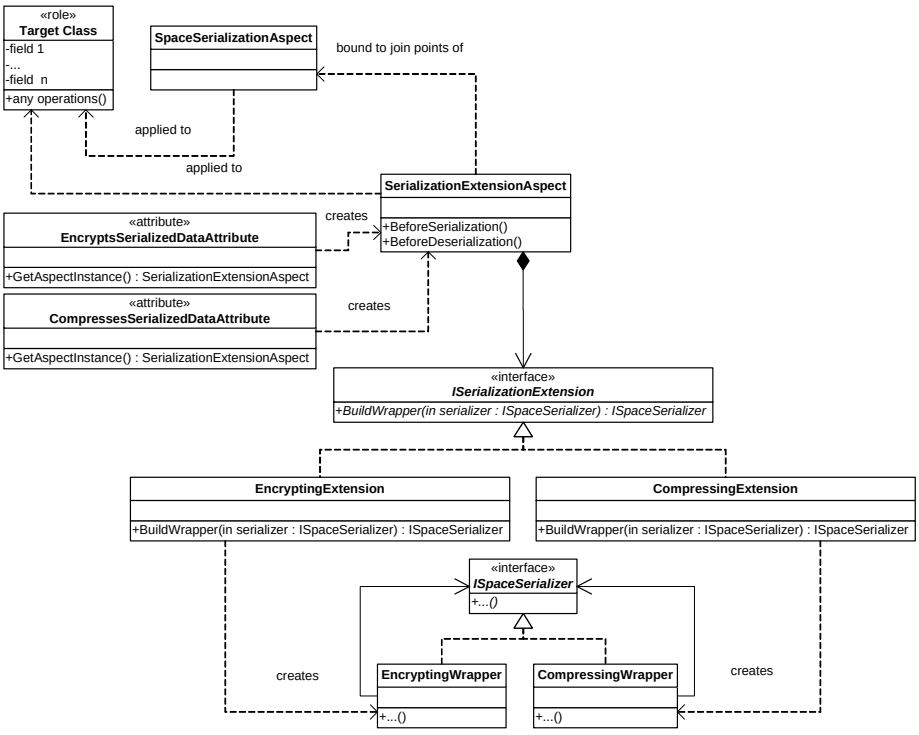

Figure 5.9: Design of the compression/encryption aspect

Evaluation

Code locality (CL): The aspect cleanly encapsulates all the concern code; crosscutting code and code for encryption/compression are cleanly encapsulated in dedicated classes and aspects.

Understandability (U): It is cleanly understandable whether the concern is applied to a target class simply by looking at the list of attributes applied to the class.

Effective code size and effort (CSE): The effort required for applying the concern to a target class is very low: only one line of code must be added

```
[EncryptedObject, CompressedObject]
[SpaceSerializable]
public class BusinessClass {
  private string sensitiveYetRedundantData;

  public string SensitiveYetRedundantData {
    get { return sensitiveYetRedundantData; }
    set { sensitiveYetRedundantData = value; }
  }
}
```

Listing 5.36: Example application of the compression and encryption aspect

	CL	U	CSE	PI	CT	CC	RC	TC	Average
Aspect-oriented	5	5	5	5	5	5	5	5	5 (excellent)

Table 5.15: Evaluation matrix for encryption and compression aspect

to the target class definition.

Performance implications (PI): No significant adverse performance implications are to be expected by the concern imlementation (apart from the costs of encrypting and compression, which however isn't caused by the concern implementation).

Changeability of target code (CT): Target code is changeable completely without affecting concern code, separate recompilation is possible.

Changeability of concern code (CC): Concern code can also be changed without affecting target code, separate recompilation is possible.

Reusability of concern code (RC): The concern is an extension of the serialization aspect and can thus be used in any scenario where the serialization aspect can be used.

Transparency of composition (TC): The aspect depends on the join points triggered by the serialization aspect (it won't do anything if these join points aren't triggered), but it has no dependencies or implications on other concerns.

Relevance for Space-Based Computing The ability of controlling access to data on the application rather than the network abstraction level is useful for every implementation of space-based computing, as is the ability of compressing an object's data before it is sent over the network. It should be noted, however, that as a particularly sophisticated space-based implementation, CORSO's successor XVSM might make development of these concerns even easier with its interceptor concept.

5.3.5 Application-Specific Concerns

There are many additional concerns which can easily be solved by employing space-based computing; and using XL-AOF, they can be cleanly encapsulated and modularized. Similar to typical design patterns [GHJV95], the concrete implementation of these concerns can often be tightly coupled to the respective use case and the application's functional concerns, but often, they can be described in a generalized and easily realizable way. In this section, we will present two such concerns, whose actual implementation can be quite specialized, but whose purpose is general enough to be discussed in this chapter. We will not give implementation and usage examples for these concerns, and we will not provide an evaluation (which would depend on the implementation), but we will give a short discussion of how these could be solved using XL-AOF for a concrete application scenario.

Space-Based Code Distribution

Goal and Usage Scenario For this concern, consider a componentized distributed application with high availability requirements ("24/7"). In such a scenario, it is extremely hard to apply updates or patches to individual components on the whole network without shutting down the application.

Using the space, it is, however, possible to distribute such patches in binary form to all the processes involved in the application simply by putting them into the space. Whenever a process instantiates an object, it could check whether a new version of the object's assembly exists in the space, and, if yes, use this code base instead of its local one. In fact, the whole application code could be put into the space in the first place, allowing any process to join in the application with only a small stub preinstalled on the computer.

Aspect-Oriented Realization This concern can be implemented by using an aspect with an *object creation* advice. The advice method intercepts the instantiation of the aspect's target types and looks up updated versions of their assemblies in the space. If a new version exists, it downloads the assembly data into a local file, loads the assembly file into memory, and instantiates the type from that assembly.

The aspect however has some impact on the object-oriented design of the application. Most importantly, the types instantiated by the aspect must be compatible with the types requested by the application, i.e. be assignable to them as defined by the rules of the .NET Common Type System. Ideally, this is solved by the application only requesting interfaces, which are then implemented by the types in the updated assembly. If the software is designed in a component-oriented way, this should not be a problem, since cleanly defined interfaces between components are one of the most important rules of that programming paradigm anyway. Implementation of this aspect with XL-AOF is possible because the framework supports aspects applied to an interface as long as the aspect provides a creation advice.

Space-Based State Versioning

Goal and Usage Scenario Usually, when an object is written into the space, any state previously held for the object is lost. This default behavior is, however, by no means mandatory—the different states of the object can e.g. be collected in a repository, allowing object edits to be tracked, differences between versions to be found, and previous states to be restored at later times.

This concern is application-dependent mainly because it cannot be generally defined when a change makes for a new version that should be kept in the repository or how much data needs to be stored. Consider a word processor, for example: the internal state of a document changes with every character being typed or deleted by the user. If every one of these changes were stored as a separate version, this would vastly worsen runtime and space efficiency of the application. Good word processors therefore go at great lengths to define what changes make for a new version and to decide whether and when the stored data consists of the whole document or a data delta only.

Aspect-Oriented Realization An easy way to implement this concern (after deciding about the versioning details) in an aspect-oriented fashion would be to have an aspect with an *after method* advice bound to the *Persist* methods introduced by the object identity aspect. The advice would determine whether the data just persisted makes for a new version and, if yes, store the data (or a delta) in a repository, which can be implemented using the directory mechanism introduced in section 5.3.1 of this chapter.

CORSO's successor XVSM makes implementation of this concern even easier with its container concept: to hold a number of different versions of an object, a bounded buffer container could be used.

5.4 Conclusion

In this chapter, we have shown five low-level concerns as well as eleven readily implemented and two application-specific high-level concerns (mostly) for space-based computing, collected in an aspect-oriented distributed concern library. The low-level concerns can be used to implement distributed data structures and other, application-specific higher-level concerns; high-level concerns either immediately add sophisticated functionality to target applications with minimal effort or extend the low-level ones with advanced behavior.

Of course, there exists an infinity of additional distributional concerns for space-based computer programs. Many of them are specific to concrete application scenarios, some are not. The AO-DCL library contains a subset of those completely generalizable concern implementations, and we believe it provides a good concern mixture, forming a solid foundation for more efficient development of CORSO-based applications. The library effectively and practically shows that aspect-oriented programming allows for clean encapsulation of concerns which wouldn't be as cleanly modularizable with traditional paradigms. Declarative programming techniques allow these encapsulations to be understandably and effortlessly applied to concrete use cases.

Chapter 6

Space-Based Programming with the DCL

The previous two chapters presented an aspect-oriented framework and a distributed concern library based on it. The former can be used for encapsulating the cross-cutting concerns of space-based applications, and the latter constitutes a declarative space interface, allowing to specify distributional intent by using $C^{\#}$ attributes.

With these prerequisites, this chapter will now provide a sort of tutorial of how XL-AOF and the AO-DCL can be used to develop distributed applications. For this, we will walk through the different steps involved in implementing the whiteboard application introduced in chapter 1, showing how the different distributional requirements can be integrated with object-oriented design in a clean and modularized way. Since the high-level aspects of the DCL are designed to be readily applicable to any application wherever needed, we will concentrate on using the low-level concerns of serialization, shared identity, notifications, and transactions to build the whiteboard program, and we will also need to add a few new aspects encapsulating error handling and security concerns.

Freshening up the application scenario, consider again the requirements specification given in chapter 1 on page 7.

> *Users should be able to log into the application from different places and collaboratively create drawings consisting of graphical shapes. Each user sees the modifications performed by other participants and can join in the drawing process, adding, removing, and manipulating shapes.*

The specification states that:

- There should be one single space data object holding attributes concerning the whole drawing rather than individual shapes;
- There should be a structured data object with information such as coordinates, size, color, and similar attributes per shape;

- Each drawing object should have links to all the shape objects comprising the drawing;
- For modifying individual shapes as well as the drawing itself, concurrency conflicts must be ruled out via transactions; and
- If participation in the drawing process is not possible due to some error, the user should be informed.

In addition, it speaks about the possibility of giving different users different privileges for modifying the drawing. In the following, this specification will be realized with CORSO, XL-AOF, and the AO-DCL.

6.1 Shared Data and Links: The Data Structure

The first thing one should think about when starting to create a space-based application is the question of data structure: what data objects will be needed in the application, and how will they be mapped to the CORSO space? Since the whole point of the distributed concern library is to provide for separation of concerns, a good way to begin is to simply start with a functional decomposition of the problem, i.e. designing the data structure exactly as it would be for a local application. Then, the mapping should be considered, applying serialization and identity aspects to the different objects in the design.

6.1.1 Functional Decomposition

In the case of the whiteboard application, the data structure consists of the drawing itself and of the shapes contained in the drawing, as indicated in the requirements specification.

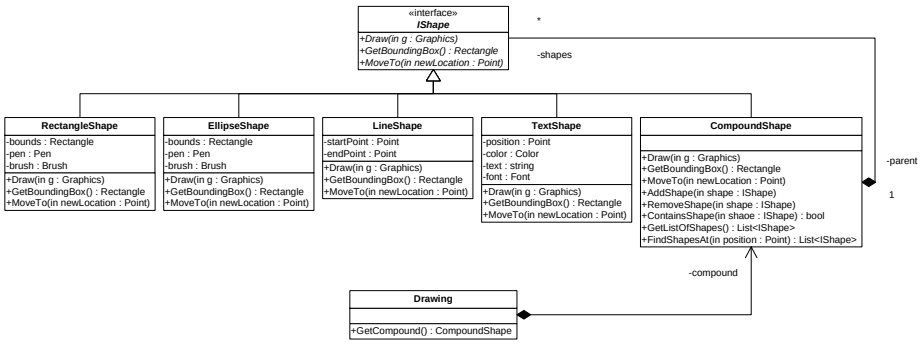

Figure 6.1: Functional decomposition of the shape data structure

For the functional decomposition, shown in figure 6.1, we start by defining an interface *IShape* providing methods that need to be implemented by all shapes likewise:

Draw is used for drawing a shape to a *Graphics* surface,

GetBoundingBox is used to compute a minimal rectangular bounding box containing the shape (this can be used to draw a border around a shape when it is selected, to calculate its center, and for performing other user interface tasks), and

MoveTo moves a shape to another position within a drawing.

This interface is implemented by a number of concrete shape classes:

RectangleShape represents a rectangular shape, defined by a bounding box, a border defined by a .NET *Pen* object, and an interior filling defined by a .NET *Brush* object;

EllipseShape represents an oval shape, equally defined by a bounding box, a border, and an interior filling;

LineShape is simply a straight line, defined by a starting point and an end point, as well as a *Pen* used for drawing;

TextShape contains user-defined text as a string, drawn in a particular color and font; and

CompoundShape implements the *Composite* design pattern [GHJV95]: it represents a group of shapes, which can together be handled just like a single shape can. In addition to the operations defined by the interface, it allows addition and removal of shapes, checking whether the compound object contains a given shape, retrieval of the contained shapes as a list, and selection of shapes from a certain screen position (which is useful for selection of shapes from a user interface).

In addition, there is also a *Drawing* class, which simply contains a reference to a root compound shape. The *Drawing* class could also contain convenience methods (such as support for cut and paste, loading and saving of shapes, printing, etc.), which are however ignored for this tutorial.

6.1.2 Mapping the Data Structure to the Space

After having the functional decomposition, the data structure must be mapped to a space coordination data structure. Starting this is simple: the root of the data structure is the *Drawing* object, which should also be an entry point, i.e. a named data object, for the space data structure.

In the functional decomposition, the *Drawing* class contains a reference to a single compound shape, which in turn contains references to other implementations of *IShape*. For mapping these to the space, there are two choices: make the shapes first-class space objects (with an identity in the space) or simply serialize them as nested structures within their parent objects.

Figures 6.2 and 6.3 illustrate this for a simple instance of the data structure, where the drawing consists of one compound shape holding a text shape and

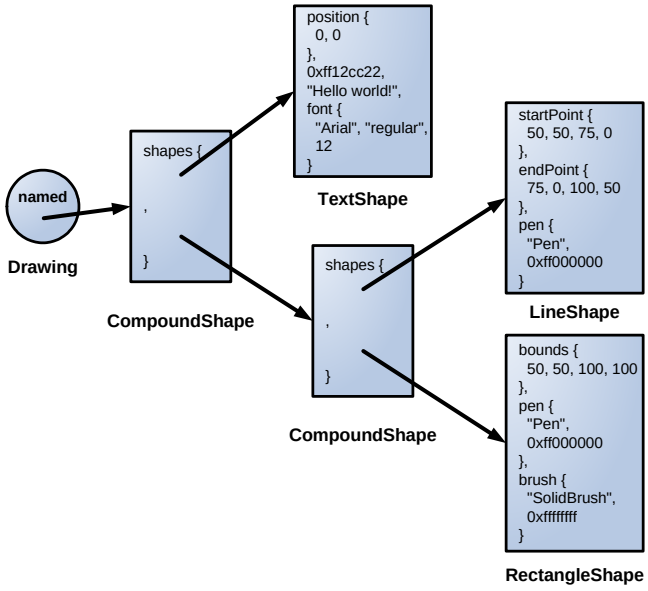

Figure 6.2: Shape data structure with dedicated space objects

a second compound shape, which in turn is comprised of a line shape and a rectangle shape. The shapes have serialized data corresponding to their fields in the functional decomposition, displayed in a simplified form in the figures. In the one case (figure 6.2), this structure is reflected in the space objects—for every shape object, there is also a corresponding space object (rectangular boxes), a named drawing object constitutes the root of the data structure. In the other case (figure 6.3, there's also a named data object for the drawing, however this time, the whole data structure is serialized into this one big space object.

Both approaches have advantages and disadvantages, as summarized in the following paragraphs.

Dedicated Space Objects

Network efficiency: With first-class shape objects in the space, it is possible to only replicate parts of the drawing to the participants, which might result in a better network utilization. On changes, only the changed objects are propagated over the network.

Identity: If shape objects have a space-based identity, this could be mapped to object reference identity. In other words, a pooling aspect could be used in order to always return the same object references for the same space objects. Methods such as *CompoundShape.RemoveShape* could then rely on the default implementation of the shapes' *Equals* methods.

Notifications: If each shape is a separate space object, the whiteboard application has to subscribe to a change notifications for every single one of them in order to react on changes made to any shape.

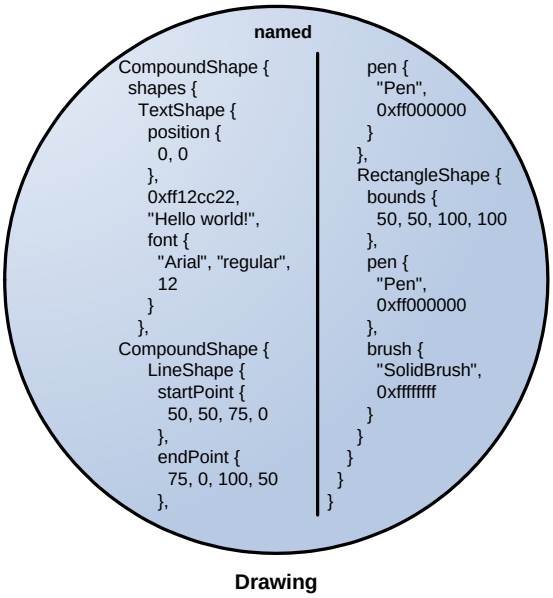

Figure 6.3: Shape data structure with one big space object

Garbage collection: If every shape object has a single space data object, it might be desirable to perform manual garbage collection rather than rely on CORSO's automatic one (which only kicks in when a process is shut down). This would mean to react on shapes being finalized on local machines and manually notify CORSO's garbage collector that the process doesn't need it any longer.

One Big Space Object

Network efficiency: In a nested structure serialization scenario, the drawing would be made of one big space object. This yields better performance when retrieving the whole drawing (e.g. when a new client joins the drawing process), but even on small changes, the whole drawing has to be transferred over the network.

Identity: With nested serialization, the serialization system would create new local shape object references whenever the drawing is read from the network. This means that the default *Equals* implementation of the shapes cannot be used for methods such as *CompoundShape.RemoverShape* or *CompoundShape.ContainsShape*. Instead, a custom implementation would be needed, checking the shape's contents rather than local object references. As a second problem, the shapes' *parent* references would have to be manually adjusted after reading a drawing from the space.

Notifications: Notifications handling would be simpler—it would be enough to simply subscribe to the change event of the drawing object in order to get notified of any changes made to any shape in the drawing.

Garbage collection: Since there is only one space object, which automatically grows and shrinks when the number of shapes changes, garbage collection would not be an issue.

In our example, we can probably ignore the issues of network efficiency and garbage collection, since the data sizes of whiteboard drawings will usually not constitute a problem. It therefore boils down to the question of easier identity management (in the case of shapes with space identities) or easier change notification management (in the case of shapes serialized as nested structures). Since it better shows how to create coordination data structures in the space using the AO-DCL, we will implement shapes with dedicated identities in this tutorial.

Extending the Data Model to a Space Data Structure

To map the functional decomposition to the space data structure given in figure 6.2, one can simply apply predefined aspects from the distributed concern library to the classes already identified.

Space-Based Identity Wanting to give every shape a space-based identity, we first apply the *space identity* aspect by means of its *SpaceIdentityAttribute*. The only implication of this aspect on the functional part of the application is that all objects in the data structure must now be created with the space object factory: instances of the *Drawing* class must be retrieved by name via *ShapeObjectFactory.GetOrCreateNamedSpaceObject<>*, and when new shapes are to be created, *ShapeObjectFactory.GetNewSpaceObject<>* must be used.

Serialization Similar to space-based identity, serialization also simply requires application of the *serialization* aspect to all the classes in the functional data structure. However, this time, it's not necessary to use the factory attribute—the space identity aspect depends on serialization, so it automatically adds the aspect to the classes.

An important issue to consider about serialization is cross-platform compatibility. When serializing an object such as a rectangular shape, for example, the serializer needs to store metadata indicating the concrete type of object being serialized. By default, it uses the full .NET type name, e.g. *Whiteboard.Shapes.RectangleShape*. However, if the whiteboard data is to be interfaced by other platforms, this type might not exist or be named differently. Therefore, one should consider creating special type identifier strings for interoperability by applying instances of *SpaceAliasAttribute* to the different classes or instances of *SpaceStructureAliasAttribute* to the assembly. Table 6.1 shows the aliases suggested for the different types used in the data structure.

Custom Serialization There are a few types in the data structure which are not directly supported by the serialization infrastructure. While classes such as *Rectangle* and *Point* are automatically parsed and serialized member by member (in alphabetical order) as substructures, this automatic behavior does not work

Type	Alias
Whiteboard.Shapes.RectangleShape	RectangleShape
Whiteboard.Shapes.EllipseShape	EllipseShape
Whiteboard.Shapes.LineShape	LineShape
Whiteboard.Shapes.TextShape	TextShape
Whiteboard.Shapes.CompoundShape	CompoundShape
System.Drawing.Rectangle	Rectangle
System.Drawing.Point	Point

Table 6.1: Aliases for type identification in cross-platform scenarios

for colors, pens, brushes, and fonts, which contain operation system handles and other data that wouldn't be valid on deserialization.

For colors, an easy way is to serialize them simply as 32-bit integers by instructing the serialization system to convert them to an *ARGB (Alpha, Red, Green, Blue)* value on serialization (and back on deserialization). This is easily achieved by using a *ConvertedForSerializationAttribute*, as explained in section 5.2.1 in the previous chapter: *[ConvertedForSerialization(typeof(int), "ToArgb", "FromArgb")]*.

For pens, brushes, and fonts, however, there is no such conversion predefined by the .NET classes. There are two choices: implement a custom conversion mechanism or implement custom serializers for these types. Since this is often the simpler way to go, we will now show how to implement a custom conversion mechanism for the *Pen* class, the idea works the same way for brushes and fonts.

As explained in the previous chapter, the *ConvertedForSerializationAttribute* takes as its parameters a substitute type which the serialization mechanism can automatically serialize. There must exist a conversion method from the source type to the substitute type (for serialization) and a conversion method from the substitute type back to the source type (for deserialization). While there is no substitute type with conversion methods for pens, one can simply create one, as illustrated in listing 6.1. This substitute type only accounts for a small subset of the capabilities of .NET pens: it stores the type of pen as a string as well as the pen's color. As previously explained, the color is configured to be serialized as an ARGB integer number.

```
public struct PenSerializedData {
  private string penType;
  [ConvertedForSerialization(typeof(int), "ToArgb", "FromArgb")]
  private Color color;

  public static PenSerializedData FromPen(Pen p) {
    PenSerializedData substitute = new PenSerializedData();
    substitute.penType = p.GetType().FullName;
    substitute.color = p.Color;
  }

  public Pen ToPen() {
    Pen pen = new Pen(color); // only supports simple pens at the moment
    Debug.Assert(pen.GetType().FullName == penType);
    return pen;
  }
}
```

Listing 6.1: Substitute type for conversion of .NET pens

With this simple substitute type, XL-AOF can now successfully serialize .NET *Pen* objects.

Operations and Transactions Since the local data structure obtained from the functional problem decomposition is now mapped to a space-based data structure via identity and serialization aspects, synchronization of local objects and their space-based counterparts should be considered next. First of all, local changes to the data structure must be synchronized to the space: calls to the *MoveTo*, *AddShape*, and *RemoveShape* methods should cause the respective shape objects to be persisted. Second, changes made by other people should be immediately displayed on the screen, so change notifications need to be used, which are dealt with in the next section. Third, the possibility of several processes changing the same shapes at the same time must be considered and guarded against.

For the first and third issues, the distributed concern library provides the *transaction handling aspect*—by applying the *TransactionalObjectAttribute* to the shape classes and the *TransactionalAttribute* to the aforementioned methods, these are automatically guarded against concurrent modifications. And without further configuration, they also cause the respective shape objects to be synchronized with the space at the beginning and at the end of the operation.

CORSO transactions are executed in an optimistic fashion, without any locking. This prevents deadlocks, but it also means that several processes can enter transactions on the same objects at the same time, which will result in only one process "winning" and being able to commit the transaction. The default transaction policy of the aspect is to retry the operation up to ten times, but in very busy scenarios, it is possible that a process does not succeed in executing the transaction, causing an error to be thrown to the caller. Due to the identity-based data structure, this will only happen when many processes are manipulating the same shape objects, so it is not very likely to happen in practice. If it does, though, it might be remedied by specifying a higher retry count in the *TransactionalAttribute*.

Change Notification To always have the local data structure objects as up to date with their space-based representations as possible—and also to make for a good collaborative experience—, the whiteboard application hosting the data model must react whenever some process connected to the space changes the drawing. It must subscribe to change notifications, applying any changes to the shape objects and updating the on-screen representation of the drawing. Therefore, the space objects must be made observable by applying the *watchable space object* aspect to them with the *WatchableSpaceObjectAttribute*. (Because this aspect depends on the identity aspect, this makes application of the *SpaceObjectAttribute* obsolete.)

For the hosting application, it is a little more complex: it must subscribe to the notification event of every single shape in the drawing and start and stop notification processing via the respective start/stop methods. This is the trade-off of the identity-based data structure mentioned before. On the other hand, the application gets the chance to optimize drawing that way if necessary: when a single object changes, it is sufficient to update (and redraw) just that object instead of refreshing the whole drawing.

When the event is raised, the event handler needs to update the shape (e.g. by calling the *Read* method added by the serialization aspect with the data

provided by the event) and then redraw the updated parts of the drawing. Since notification events come asynchronously, in parallel to normal program flow, this requires thorough synchronization. With Windows applications, this is quite easy, however, since all such applications follow a message pumping model with a single *UI thread* per window or group of windows on which all window manipulations must be performed [Sel04]. For the whiteboard, it is therefore sufficient to post an update message to the hosting application's message pump (a standard procedure in .NET UI programming) and do the updating from the UI thread to ensure proper synchronization.

Pooling The most important advantage of choosing a data structure with space-based identities is the possibility of tracking local object references in a pool, reusing the same reference whenever a certain space object is needed. For this, we presented the *pooling* high-level aspect in chapter 5, and indeed it's enough to simply apply the *PooledAttribute* to the shape classes in order to make the space object factory handle pooling correctly. As a result, there will be at most one local object reference per space data object, no matter how often its parent object is deserialized. As indicated before, this makes the default implementations of the *Equals* method, which is used by the compound shape's *ContainsShape* and *RemoveShape* methods, work correctly; and it even results in the *parent* back reference of the shapes being deserialized correctly.

Garbage Collection As mentioned before, garbage collection in the space might be an issue with large drawings and long-running processes. Since the automatic garbage collection supported by CORSO is based on process reference counting and therefore only kicks in when a process disconnects, it might be useful to manually free space data objects that aren't needed any longer. The *lifetime tracking* high-level aspect given in the previous chapter helps with that by releasing a process' reference to a space data object when its local representation gets claimed by the .NET garbage collector. Since every space data object has at most one local representation due to pooling, the lifetime tracking aspect can be used by applying its factory attribute to the shape classes.

6.2 Integrating into the Application

Now, by simple application of six pre-defined aspects to the data structure constructed through functional decomposition, we have achieved full mapping to a space-based data structure. For most aspects this only needed one attribute to be applied to the data structure classes, only serialization required a little work for the specialized graphical types. The space-based data structure can now be integrated into an actual graphical whiteboard program with a graphical user interface, e.g. into one with a design similar to the one in figure 6.4.

In the figure, there is a main window, which is a standard .NET Windows Form, comprising a tool bar and a drawing canvas. The canvas is a .NET control constituting the main drawing area and hosts the actual *Drawing* object, while the tool bar contains a number of tools the user can select to change the way drawing is performed.

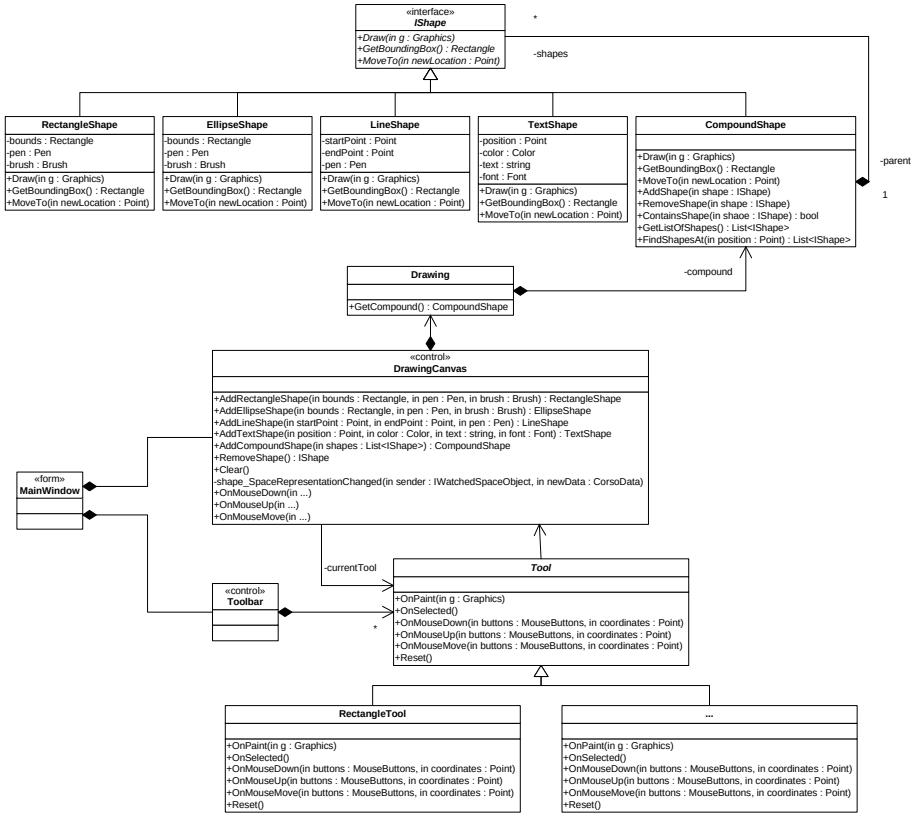

Figure 6.4: Whiteboard application sample integrating the data structure

The canvas is responsible for loading the drawing from the space (using the space object factory's methods) at application startup time, it delegates mouse commands entered by the user to the active tool, and it holds a number of methods the tools can use for creating and adding new shapes. Via these methods, the canvas can ensure that it is always subscribed to all change notification events, which are handled by the *shape_ SpaceRepresentationChanged* event handler by scheduling an update on the main UI thread, as previously explained.

The tools, which are managed via the tool bar and selected into the drawing canvas' *currentTool* property, are derived from a common abstract base class, which defines an interface via which a tool can paint itself, initialize when being selected, reset itself (e.g. when the user presses *Escape*), and react to the mouse commands coming from the canvas. The figure only shows a concrete tool class for drawing rectangles, but similar subclasses (indicated by the ... tool) exist for ellipses, lines, text, and for grouping shapes together into a compound shape.

This design was implemented in a whiteboard program prototyped for "Attributes &Co – Collaborative Applications with Declarative Shared Objects" [eKS05a]; two screen shots of the application are given in figure 6.5.

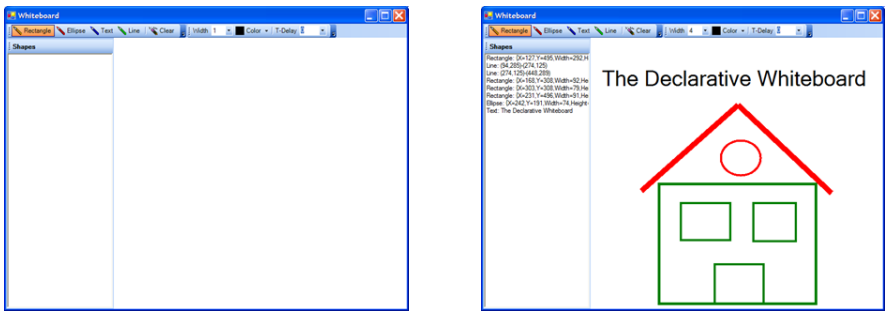

Figure 6.5: Sample user interface hosting the data structure

6.3 Fallback: Informing the User of Errors

Although we have now described all the steps necessary for creating a completely functional collaborative space-based application using the distributed concern library, there is an important point yet to consider: error handling. While CORSO manages to mask many kinds of errors, there are still some that must be explicitly handled by the application. For example, the connection to the CORSO kernel could break (causing a *CorsoException* to be thrown), the connection between kernels could break (causing space operations to time out and *CorsoTimeoutExceptions* to be thrown), somebody could corrupt the data structure in the space (resulting in *CorsoDataExceptions* being thrown), and starvation might occur, as indicated in the paragraph on transactional operations.

In this tutorial chapter, we will not implement a sophisticated error handling mechanism with nice user-interface integration. Instead, we will simply show how to write an aspect handling such exceptions by showing a message box to the user. Nevertheless, the same aspect-oriented mechanisms could equally well be used to implement user-friendly solutions, which is however out of scope for this thesis.

Although we could use the UI error handling aspect, which is already part of the distributed concern library (see section 5.3.3), that one is not extensible and more suitable for rapid prototyping scenarios rather than real applications. If we wanted to upgrade this message box-based solution into a sophisticated one later on, we would need a custom implementation anyway.

We therefore write a custom aspect to encapsulate error handling code in a cleanly modularized way. The aspect advises XL-AOF's *after exception* join point, which applies whenever an exception gets thrown from a (virtual) method, and we do not use a filtering pointcut expression, which results in the advice being executed for *any* uncaught exception on *any* virtual method on the target type. By applying this aspect on the shape classes, all errors not handled within the data structure get propagated to the user, who can then decide what to do next. The aspect code is shown in listing 6.2. It synchronizes the actual error dialog to the UI thread using *Form.ActiveForm.Invoke*, because message boxes may not be shown from any other thread (while errors can occur on these threads). Since the advice chooses to swallow the exception, it needs to return a

"default" value; i.e. *0* for primitive types, an empty value for value types, or *null* for reference types (and void methods). To obtain this value, a helper method provided by XL-AOF's *AspectEnvironment* is used.

```
public class ErrorUIAspect {
  [Advice(typeof(AfterException))]
  public object TellUser(MethodInfo method, Exception exception) {
    string message = string.Format("An error of type {0} occurred in " +
      "method {1}. Exception text: {2}", exception.GetType().Name,
      method.Name, exception.ToString());
    Form.ActiveForm.Invoke(delegate { MessageBox.Show("Error", message,
      MessageBoxButtons.OK); });

    // return a default value for the respective return type
    Type returnType = method.ReturnType;
    return AspectEnvironment.GetDefaultValue(method.ReturnType);
  }
}
```

Listing 6.2: Custom, cleanly modularized error handling

6.4 Security: Different Roles for Different Users

Now, nearly all of the distributional requirements stated for the whiteboard application have been integrated into the data structure obtained from the functional problem decomposition in a clean, modularized, and mostly very simple and efficient way. The only requirement that is still missing is that of security checking—having different roles for different users. In particular, we introduce the following security system:

- When starting the whiteboard application, users need to log in.
- A security database determines which roles are assigned to each user. For example, users can be administrators (which can remove and modify any shape) or default users (which can remove and modify only those shapes they created).
- Every shape object has a token associated with it which identifies the user it was created by.
- Every method on a shape object checks with the security database if the user currently logged in is allowed to perform the operation, considering the user's identity and role as well as the shape's creator.

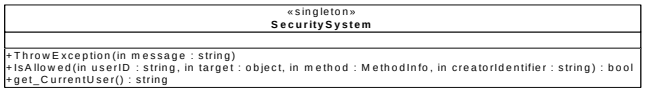

Figure 6.6: Security database interface for the whiteboard application

For this concern, there is no predefined implementation in the distributed concern library (although we've already discussed a similar concern in chapter 4, section 4.4.4), so we need to manually implement the security system. The first issue, login functionality, is not really a cross-cutting concern, since it doesn't

cut through any design. It can be cleanly encapsulated as a graphical user interface component shown at application startup time. The security database for the second requirement can be implemented as a cleanly encapsulated object-oriented component, e.g. as illustrated in figure 6.6. In the figure, there is just one singleton class representing the API of the security component. It provides mechanisms for retrieving the currently logged in user, checking whether an operation on a target with a certain creator is allowed, and throwing security exceptions in case of violations of the secutiry rules.

The third and fourth requirements, however, need to be integrated into the functional data structure. This makes them clearly cross-cutting, so they are best implemented as aspects. In fact, we will use a single aspect holding the creator identifier for its target shape object (which make it necessary for the aspect to be instantiated on a per-object basis). The aspect advises the method calls performed on the shape, checking for every method execution whether the operation is allowed. The code in listing 6.3 shows how to implement such an aspect with one data member for the identifier and an advice method bound to the *before method* join points of arbitrary method calls on the target class. The advice delegates to the security system interface, passing it the necessary context information and throwing an exception if the system decides that the operation is not allowed. If the security system authorizes the operation, no exception is thrown and execution of the method continues.

```
class SecurityAspect {
  private string creatorIdentifier;

  public SecurityAspect() {
    creatorIdentifier = SecuritySystem.CurrentUser;
  }

  [Advice(typeof(BeforeMethod))]
  public void CheckPermissionsBeforeMethod(object target, MethodInfo method) {
    if (!SecuritySystem.SecurityDatabase.IsAllowed(SecuritySystem.CurrentUser,
      target, method, creatorIdentifier)) {
      SecuritySystem.ThrowException("The current user is not allowed to" +
        " perform the operation " + method + " on the object of type " +
        typeof(target).FullName + ".");
    }
  }

  [Advice(typeof(BeforeObjectSerialization))]
  [Advice(typeof(BeforeObjectDeserialization))]
  public void AddMemberToSerializationProcess(ICollection<ReflectedField> fields)
    {
    fields.Add(AspectEnvironment.FieldOf<string>(this, "creatorIdentifier"));
  }
}
```

Listing 6.3: Security aspect implementation for the whiteboard application

The aspect initializes the creator of its target shape to the user currently logged in on construction. However, this default value will not be correct when an existing shape is loaded from the space; similarly, the owner identifier must also be persisted when the shape is written to the space in order to communicate it to others. Therefore, the aspect also contains advice for the *before object serialization* and *before object deserialization* join points of its target objects. These are are triggered by the serialization aspect and allow the security aspect to process additional data when serializing or deserializing a shape object. The

additional data (respectively the field which is to receive the data) must be provided in form of *ReflectedField<>* objects, which can be created easily using the *AspectEnvironment* reflective class. Since the logic to be performed for serialization and deserialization process is the same, the same advice method can be used for both join point kinds.

This aspect can be applied to the shape classes using the predefined per-object factory attribute or by defining a custom one.

6.5 Wrapping Up

In this chapter we have led through the process of creating the space-based part of a distributed application with the help of XL-AOF and the aspect-oriented distributed concern library. We took a straight-forward approach consisting of the following steps:

- Create a functional decomposition, as you would without distributional concerns.
- Plan how to map the data structure to the space, i.e. which objects need to be serializable and which additionally need space-based identities. This mapping is important, since it contributes to the efficiency and ease of use of the resulting shared data structure. In our example, we had to balance the advantages regarding network efficiency and reference identity of the identity-based mapping versus the notification handling and garbage collection advantages of the more monolithic variant. In out case, the mapping did not require any changes on the functional decomposition, but in general, slight adaption might be required.
- Map the data structure to the space by applying the predefined AO-DCL aspects to the objects of the data structure. If you use any special classes (such as the fonts and pens used in our example), specialized serialization might be needed. This is most easily implemented by adding substitute types as we showed for the *Pen* class.
- Identify the operations that should be transactionally safe with regards to the space; usually this involves those methods performing any changes on objects with space-based identity. You can also include convenience methods that group single actions to bigger transactions if this helps the user.
- Apply the predefined transaction aspect to the respective objects and tag the methods accordingly.
- Find the objects with space-based identity you want to use change notifications upon and apply the predefined aspect from the AO-DCL. Don't forget to manually start and stop event notification when using the data structure in a program.

- Identify and apply any other, higher level concerns from the concern library that apply to your scenario. In our case, we used the pooling aspect to get reference identity for shared objects and the lifetime tracking aspect in order to aid the middleware layer in memory management.
- Consider fallback requirements and identify the places in the program where they should be handled. You can write a custom aspect using XL-AOF if you want to encapsulate cross-cutting error handling code.
- Consider any other cross-cutting requirements for which there are no predefined concerns in the library (in our example that was the security concern). Write custom aspects for each of them, which shouldn't be much work with the functionality provided by XL-AOF.

By following this "recipe", space-enabled applications can be developed in a highly efficient way, with cleanly modularized, easily understandable results. On the one hand, this is due to the highly declarative distributed concern library and its pre-built solutions for the usual concerns of space-based applications, which can easily and readably be integrated into the functional problem decomposition. On the other hand, it is made possible by XL-AOF, which makes the reusable encapsulations of the library possible in the first place and also provides a way to extend the program by additional aspects for concerns not predefined.

As a side note, this list of instructions does not necessarily apply when space-enabling one small part of an application, for example in order to achieve distributed monitoring. In such cases, it is often enough to just apply one or a few high-level library aspects to the respective classes without having to think about space data structures, mappings, etc.

Chapter 7

Summary and Evaluation

The previous chapters introduced an AOP infrastructure based on runtime-generated subclass proxies. They described a lightweight and extensible aspect-oriented framework built on top of it, which enabled the creation of a declarative distributed coordination concern library for space-based computing. All this was under the single common goal of achieving an *efficient* development experience for space-based applications.

This goal was presented and detailed in the introduction chapter of this thesis, which described current interfaces to space-based middleware layers to be too complex—the advantages of the space-based computing paradigm notwithstanding. We attributes this to the imperative object-oriented programming paradigm, whose rules these interfaces have to adhere to, and identified declarative techniques to be a good alternative for space-based interfaces:

> *We believe that a successful approach should provide [...] high-level declarative access to middleware services from within the context of already commercially successful* general-purpose *programming languages such as C# or Java. [...] A declarative version could remedy [the problem] by providing a better interface to the network abstraction, subjectively reducing complexity to that minimal amount which is essential due to the program requirements.* Introduction chapter

While modern object-oriented programming languages have some built-in support for declarativity through customized tagging of program elements (extensible metadata), declarativity alone won't solve the problems of complex interfaces. In addition, programmers need mechanisms allowing them to modularize the implementations and algorithms behind the declarations. We found aspect-orientation to be a programming paradigm offering these mechanisms through so-called *aspects*:

> *The aspects translate the intents specified by the programmer to algorithms executed at runtime, enabling goal-directed code to be written although algorithms are executed.* Introduction chapter

In chapter 2, this work therefore investigated the state-of-the-art of these technologies and especially of the idea of achieving efficient distributed application development through combination of space-based computing, declarativity, and aspect-oriented programming. This yielded a feature matrix (see table 2.5 on page 42), which showed that an implementation of this idea didn't exist at the moment.

As a result, this thesis constitutes the first existing attempt at creating and analyzing an integrative solution of space-based distributed computing, declarativity in object-oriented languages, and lightweight, easily adoptable aspect-oriented programming, all with the final goal of efficient space-based application development. In this chapter, it will be investigated whether this final goal has been fulfilled by pulling together the evaluation parts given in the course of this thesis, summarizing their results and putting them into context.

7.1 The Suitable AOP Infrastructure with Great Adoptability Potential

In chapter 3, the first step to achieving a better space-based programming experience was taken with an AOP infrastructure for aspect-oriented programming. We identified runtime-generated subclass proxies as a suitable weaving technology, and provided an analysis of the concept, finding two important characteristics:

- The runtime-generated subclass proxy approach is definitely limited what regards aspect-oriented features.
- It is, however, highly advantageous what regards psychological and adoptability properties.

The first point refers to the join point model limitations of a proxy-based approach. Subclass proxies lack features typically found in code weaving approaches: subclass proxies can simply not intercept class construction, field access, nonvirtual method execution, method calls (as opposed to executions), nonescaping exceptions, or single control statements, and thus cannot expose those features as aspect-oriented join points. In addition, client code of an aspect-oriented framework built on subclass proxies needs to instantiate objects through a dedicated factory in order to have the proxies correctly instantiated.

What does that mean in the context of space-based computing, though? As was shown in the later chapters on implementing space-based concerns in an aspect-oriented way (chapters 5 and 6), it doesn't have much implication: the join points not supported aren't needed for this kind of concerns at all. When space objects are instantiated, a factory is mandatory in any case, because space object IDs need to be looked up and the locally created instances have to be synchronized with space data.

There is only one restriction posed by this infrastructure which remains in the context of space-based computing: non-virtual method executions cannot be

join points. In effect, this means that all methods guarded against concurrency problems and errors or otherwise influenced by aspects need to be made virtual.

The second characteristic mentioned, subclass proxies standing out against other AOP approaches regarding psychological factors, has more implications, though. Common approaches face adoptability problems such as

- Aspect-oriented programs becoming incomprehensible,
- Existing tools becoming incompatible with the approach, and
- Infrastructure not being available due to its complexity.

These problems were shown to be ameliorated by the choice of runtime-generated subclass proxies as an AOP infrastructure. This is important, because efficient space-based application development must not be hindered by the typical adoptability problems of aspect-oriented solutions.

With these conceptual evaluation results, a performance evaluation of the infrastructure approach was conducted. Of course, dynamic mechanisms such as runtime-generated subclass proxies can never be faster at runtime than compile-time approaches are, but the question was how much performance penalty subclass proxies really introduce.

First, we analyzed the performance of object creation with two existing implementations of the approach. These implementations generated the proxy classes when the target classes were instantiated for the first time, so a slowdown was expected. Emitting a new class and instantiating it is much more complex than simply allocating memory for an instance of an existing class, so the measured performance hit was high indeed: instantiation needed about 12-37ms with proxy generation as opposed to about as many nanoseconds with an ordinary allocation. This is a relative factor of one million, and seeing only the isolated numbers, it seems catastrophic at first.

On the other hand, the absolute numbers should be regarded: several milliseconds for the first instantiation of a class are not really a problem. Even if an application instantiated fifty different classes all at the same first time—which is a very improbable situation—, this would only yield about a second delay. Typically, there will be fewer classes, and they will not all be instantiated at the same time, so the resulting delays won't even be noticeable in user interaction [Saf06]. And since subsequent instantiations do not require repeat class generation, class generation time should not be an issue for all practical purposes.

Next, the performance penalty induced on methods intercepted for join point handling was measured and analyzed. This is typically more important than object creation, because method invocations usually occur much more often in programs than object instantiations do[1].

[1] This is obvious considered that in order to work with an object in object-oriented sense (i.e. an encapsulation of data and behavior) at least *two* method calls are necessary for every object instance: the constructor call and the method holding the object's behavior. For such objects there will be at least twice as many method calls as object instantiations. This point is not valid, of course, for pure data containers without behavior.

An analysis showed that this penalty can be in the order of one non-virtual method call if the aspect needs to call back to the original method. If it doesn't, there is no penalty at all—the virtual call made by the calling code is simply automatically redirected to the subclass proxy.

In praxis, however, existing implementations of the infrastructure have chosen not to implement this highly performant interception mechanism. Instead, they go via delegates or even via Reflection to invoke the original method, which is somewhat more expensive than a non-virtual method call. We have found the reason for this to be that more efficiency simply isn't necessary: with methods holding nonempty (and nontrivial) bodies, it doesn't matter whether a method is invoked via an ordinary call instruction or via a delegate—performance bottlenecks stem from *within* the methods. This is even more so in the context of distributed applications: in these, the bottleneck is in data transfer, not in the time needed to invoke a method.

Summarizing, feature-wise and performance-wise, runtime-generated subclass proxies are at least a *sufficient* means for implementing an aspect-oriented toolkit. As was seen in subsequent chapters, they provide everything needed to facilitate efficient space-based application development. Regarding their adoptability potential, they are even a *great* infrastructure.

7.2 The Aspect-Oriented Framework Well-Suited for Space-Based Development

The next step was to build an aspect-oriented framework in chapter 4. The framework would later be used to encapsulate the cross-cutting code behind the declarative language binding suggested in chapter 1, and for this, three important features were identified: light weight, extensibility, and adoptability.

In section 4.1, a tool was defined to be of light weight when it is cleanly integrated into the tool chain, is naturally integrated into a mainstream programming language, weaves in a noninvasive way, and allows one to decide at runtime whether to use it or not. A tool was said to be extensible if new modularized cross-cutting concerns can easily be added even though they were not considered at design or implementation time of the tool. And adoptability was stated to require that an aspect language is easy to learn, that the tool has a simple and efficient way of build and application configuration, that it relies on well-known and easily understandable infrastructural concepts, that it does not need complicated or hard-to-use build or weaving tools, and that it does not break any mainstream tools typically used for building the system.

In the course of the chapter, an aspect-oriented tool was developed, which indeed is lightweight: as a framework, it is cleanly integrated into the tool chain, as it does not introduce any additional tools at all; it is simply a library. It is naturally integrated into mainstream .NET programming languages with its aspect language being based on the declarative metadata mechanisms defined by the .NET platform standard. It is built on the subclass proxy infrastructure and thus weaves in a noninvasive way, and as a factory-based framework dynamically configurable, it allows one to decide at runtime whether to use it or not. In these

terms, it is different from other aspect-oriented toolkits currently in use (see also chapter 2).

Its aspect language provides general aspect-oriented features, which allows new cross-cutting concerns to be implemented easily, even if the concerns have not been anticipated at framework development time. This can be seen from the later chapters 5 and 6, where the aspect language was used to implement all the different general-purpose concerns as well as one special-purpose requirement in a cleanly modularized fashion. This distinguishes the aspect-oriented framework from the existing non-AOP application frameworks or application containers for distributed (or even space-based) computing.

Regarding adoptability, the framework was built on top of the subclass proxy infrastructure, inheriting the positive adoptability properties of that technology. Via the infrastructure, the framework only relies on well-known object-oriented concepts. It does not need any additional build or weaving tools, and it does not break any existing mainstream tools typically used for building .NET applications. The framework does not require any build configuration, and application configuration can be intuitively done via declarative attributes, or, if the power is needed, from imperative code.

The aspect language was modeled as intuitively as possible, and to prove this, adoptability properties for every single language feature were collected. Naturally, a full AOP tool has some complex properties: we identified 16 points that could negatively influence adoption of the framework because of complicated and advanced features or because of the infrastructure's limitations; each of those remains in the framework with a good rationale.

On the other hand, 40 positive adoptability issues were identified, features of simplicity, which make our framework very easy to use, or features of expressive power, which make problems very easy to solve. We were able to illustrate the comprehensiveness of our framework by providing a tutorial consisting of five general-purpose sample aspects, which illustrated both extensibility and adoptability features of the framework: logging, parameter checking, call privileges, atomicity, and property change notification.

Each of the samples was informally analyzed and compared to a classical, object-oriented implementation of the same concerns. The results of this analysis are illustrated in table 7.1: a plus sign indicates that the aspect-oriented version improves the code for the respective property, a minus sign indicates that the object-oriented version yielded better results.

	Log.	**Param.**	**Privil.**	**Atom.**	**Notif.**
Code locality/understandability	+	+	+	+	+
Reusability of aspect	+	+	+	+	+
Scalability of aspect	+	+	+	+	+
Changeability of class	+	+	+	+	+
Changeability of aspect	+	+	+	+	+
Declarative documentation		+	+	+	
Reusability of class	+				+
Configurability	+				
Code size/effort		+	-	-	-
CFlow needed	-				
Object factory/virtual methods	-	-	-	-	-

Table 7.1: Sample aspect evaluation

As can be seen from the table, all the sample concerns profited from improved code locality and understandability, aspect reusability and scalability, as well as aspect and class changeability in their aspect-oriented implementation. Most also profited from the declarative documentation obtained via aspect application through declarative attributes. Two aspects made the target classes better reusable and one aspect made the concern better configurable.

On the other hand, all aspect-oriented solutions of course suffer from the restrictions imposed by the subclass proxy approach—they need to use object factories for instantiation, and methods to be used as join points must be virtual. One aspect needs to make use of a control flow-analyzing pointcut, which is somewhat more complicated than a straight-forward object-oriented solution.

Three of the samples need greater code size and programming effort for the aspect-oriented solution than for a straight-forward one, which seems contrary to the actual goals of aspect-oriented programming. The reason is simple: each of the samples showed the cross-cutting concerns in the context of just one class, so the concerns didn't really cross-cut a lot. Aspects are superior in code size and programming effort only in cases where the concerns cross-cut a lot, involving many different classes. This can be seen with the only sample concern cross-cutting heavily on a method level: for the parameter checking concern, the aspect was superior even in the simple sample case.

How does this all fit into the context of simplifying space-based application development? In a later chapter, the declarative distributed concern library was built on top of the aspect-oriented framework. And just like the samples profited from better code locality and understandability, reusability, scalability, changeability, and documentation, the distributional concerns later implemented in the library profited from these properties. And, since these aspects were designed to be implemented once, but used often, they of course also profited from the better code size/effort properties brought by aspect-orientation. Together with its good adoptability features, XL-AOF is therefore perfectly well-suited for building an efficient space-based programming interface upon. And due to its focus on extensibility, programmers can easily augment the library with their own modularizations of cross-cutting concerns when needed.

7.3 The Distributed Concern Library for Highly Improved Code Quality

In chapter 5, the declarative language binding motivated in the first chapters was finally implemented. The most important low-level concerns dealt with by the CORSO middleware, as a mature representative of the space-based paradigm, were taken and encapsulated into reusable, cleanly separated, and declaratively applicable modules with the help of the aspect-oriented framework developed in chapter 4. Then, a number of useful high-level concerns were implemented, also in an aspect-oriented fashion. Together, these concern implementations form an aspect-oriented and declarative distributed concern library (AO-DCL), whose goal is to provide an efficient way of developing space-based distributed software. And while this library is implemented on top of CORSO, we analyzed

the concerns it contained to be of relevance also to XVSM, CORSO's successor, and space-based computing in general.

To prove that the concern library really leads to efficient space-based programming, each concern implementation was evaluated by analyzing its source code and giving it score points for different static code quality properties. The evaluation schema for this process was taken and extended from Hannemann and Kiczales [HK02] and Pawlak et al [PSR05], and the source code was characterized regarding the following properties, with possible scores ranging between one point (catastrophic) and five points (excellent) per item:

- Code locality, modularity, and separation of concerns;
- Effective code size and effort needed for integration of the concern in an application;
- Performance implications caused by the concern;
- Target code changeability;
- Concern code changeability;
- Concern code reusability; and
- Transparency of composition.

For each concern implementation, an overall score was calculated by taking the average of the property score points. To show that the aspect-oriented implementation is actually superior to the classic imperative approach, object-oriented variants of the low-level concerns were presented and evaluated as well. Table 7.2 shows the scores of the low-level concerns in both aspect-oriented and object-oriented implementation.

Concern	Object-Oriented	Aspect-Oriented
Serialization	2.375	4.875
Space-Based Identity	2.75	5
Notification	3.25	5
Transactional Safety	1.875	4.75

Table 7.2: Scores of low-level concern implementations

Analyzing these figures of the most basic concerns, which are the foundations of all CORSO applications, there were no excellent, not even good implementations with the classic object-oriented API. The best score is a mediocre 3.25 (notification handling), the worst is 1.875 (transaction handling), which is quite poor from a code quality standpoint.

On the other hand, when built on top of the aspect-oriented framework, half of the concerns were assigned an excellent 5 points, and the "worst" score is 4.75, which is still very good.

These numbers support the main hypothesis of our work: declarative and aspect-oriented techniques can be used to provide a *more efficient* space-based application development experience than current standard practices do.

And then, they even allowed to go a step further—using the aspect-oriented framework, we were able to define clean encapsulations of high-level concerns. These comprise self-contained units which can be applied to a program to immediately make use of space-enabled techniques as well as building blocks for sophisticated space-based applications. They provide out-of-the-box solutions to issues such as distributed monitoring and caching, as well as pooling or lifetime tracking problems. Issues like these cannot be reusably encapsulated using object-orientation, they can only be described as design patterns, which have to be implemented every time they need to be used, as described by Gamma et al [GHJV95] and Hannemann and Kiczales [HK02]. Aspect-oriented technology, on the other hand, allowed them to be encapsulated and included into a concern library.

Table 7.3 shows the score summary of their evaluations. Since these concern implementations highly depend on the low-level aspects and cannot be reusably encapsulated with purely object-oriented techniques, they have not been separately compared with object-oriented variants.

Concern	Aspect-Oriented Implementation
Mirroring	4.5
Tracing	4.5
Heartbeat	4.875
Local Caching, Distributed Results Cache	4.875
Error handling	4.75
Auto-refreshed object	4.875
Pooling	4.875
Lifetime tracking	4.75
Up-to-dateness service level agreement	5
Encryption and compression	5

Table 7.3: Scores of high-level concern implementations

The table clearly shows that aspect-orientation leads to good software quality in the field of space-based computing: there are two concern implementations which had excellent scores (5 points), and no concern had less then 4.5—a very good result.

The final evidence for the effectiveness of our approach towards efficient space-based application development was given in chapter 6, where a methodology for implementing space-based applications with the declarative concern library and the aspect-oriented framework was presented. Based on the example of a collaborative whiteboard application, the chapter provided a "cookbook", showing that with the help of the library, most requirements of distribution could be solved simply by declaratively specifying intent: through mere application of declarative attributes. The efficient declarative language binding demanded in the introduction chapter of the thesis has thus been realized.

Chapter 8

Conclusion

Now and in the last 40 years, a vast number software projects have been canceled due to their complexity. While the essential part of this complexity is inherent to the problem and cannot be coped with, the accidental part can—by the use of the right tools. Distributed applications are more difficult to develop than non-distributed computer programs because their level of essential complexity is higher; they have to deal with a number of additional requirements when compared to local ones. It's therefore especially vital to employ the right tools for developing distributed applications.

The most important tools, middleware systems, help to cope with the complexity of distributed applications by abstracting as many details as possible, eliminating accidental difficulties. The space-based computing paradigm is a particularly good existing abstraction, providing a natural, data-centered communication model well-suited for many applications.

However, as we have illustrated throughout this work, the object-oriented interfaces space-based middlewares have traditionally provided to programmers are often not intuitive and hinder an efficient development process. In the context of space-based applications, the imperative object-oriented programming paradigm, which forces developers to explicitly formulate algorithms and implementations of their distributional intentions, directly exposes the programmer to accidental difficulties.

This thesis proposed a solution to this problem, a way to reduce the accidental difficulties of space APIs and a way to provide an efficient space-based development experience: it described a *declarative* interface to the space-based middleware, a way to employ the declarative metadata mechanisms of modern object-oriented programming languages for expressing intents of space-based computing. In hindsight, it seems that space-based computing was a particularly good basis for the realization of this idea, because its data-centered approach, which often results in a high correlation of space-based coordination data structures and their local object-oriented counterparts, lent itself to a goal-directed, intentional, and declarative way of programming.

The space-based concerns expressable via declarative metadata were, however, inherently cross-cutting and therefore effectively resisted modularization through

object-oriented means. And so, the thesis adopted *aspect-oriented* methodology in order to successfully encapsulate these concerns into reusable and extensible modules.

An aspect-oriented weaving infrastructure for .NET was designed, a technology that allowed to implement aspect-oriented programming on the purely object-oriented .NET platform. As a particularly lightweight approach, we chose runtime-generated subclass proxies for this infrastructure, a dynamic weaving mechanism purely based on object-oriented mechanisms and runtime code generation, both well-supported by .NET. Based on this infrastructure, the aspect-oriented framework XL-AOF was devised and implemented[1]. Due to the novel factory attribute approach, where declarative attributes are used to instantiate aspects on target classes and objects, the framework was designed for declarative configuration from the start. It is well-integrated into the .NET platform, and even allows for dynamic aspect configuration for respective use cases.

Finally, the actual declarative space interface was implemented with the aspect-oriented framework: the declarative distributed concern library AO-DCL. AO-DCL is a catalog of preimplemented, reusable aspects for many requirements in applications based on the CORSO middleware. It comprises realizations of the most important low-level space concerns supported by CORSO—shared objects with serialization and identity, transactional operations, and notifications—as well as a number of high-level components. The resulting declarative middleware interface confirms the idea of it being more efficiently employable than the original object-oriented API.

In a dedicated chapter, we showed the process involved in creating a space-based application with the AO-DCL by implementing the distributed data structure of a collaborative whiteboard application. We demonstrated that this process mostly consisted of applying declarative attributes on the application's functionally decomposed data structure, and we also illustrated how XL-AOF could be used to implement a special issue not preimplemented in the AO-DCL. We generalized the process shown in that chapter to provide a "recipe", a list of instructions for implementing distributed applications with the AO-DCL.

To further support our claims and show that this thesis has indeed managed to reach the goal of providing efficient space-based computing through declarative and aspect-oriented techniques, we lastly summarized and put into context the evaluations done throughout this thesis in a separate evaluation chapter. With its results, we can now conclude that we indeed managed to replace the original space-based application architecture with a new and improved one: incorporating the technologies presented in this thesis, programmers can now fully leverage space-based computing with the declarative AO-DCL and encapsulate any further cross-cutting concerns with the powerful, yet lightweight XL-AOF. And it should also not remain unmentioned that XL-AOF constitutes a great, lightweight, and easily adoptable aspect-oriented framework for *general* .NET-based application development as well.

[1] A full implementation of the framework is available and can be obtained via the author.

8.1 Future Work

Although the thesis ends with this conclusion, the work it describes prepares the ground for future research and implementation work. First of all, at the time of this writing, the first open source implementation of XVSM, the new sophisticated space-based middleware, is being created. While many concepts of XVSM are different from those of CORSO, the fact of space-based programming being full of cross-cutting concerns remains—a distributed concern library should therefore be created for XVSM as well. Second, many ideas remain yet unresearched what regards the combination of aspects and spaces; for example the dynamic distribution of aspects over a space is facilitated by XL-AOF's means for dynamic weaving, but it is still a matter of some complexity. Also, broad usability studies could further strengthen this thesis' points with empirical data for selected aspects. Third, as the .NET platform evolves, new possibilities for AOP infrastructures and language support will come up that should be analyzed and used to extend XL-AOF whenever fitting. And finally, development of debugging visualizers, aspect browsers, and similar IDE extensions will all contribute to make aspect-oriented programming and thus space-based application development even more efficient.

Appendix A

.NET &Co—CORSO's .NET Language Binding

This appendix shows the public APIs of most of the classes and interfaces constituting CORSO's .NET &Co language binding for reference purposes. The shown code was automatically generated by the free *Reflector* tool for .NET.

The semantics of the shown classes and methods are shortly described in section 2.1.2, for a more detailed and more complete reference see the *Corso .NET &Co Documentation* [Tec04c].

A.1 CorsoConnection Class

```
public class CorsoConnection
{
  // Methods
  public CorsoConnection();
  public void AddService(string typeCreator, string typeName, string serviceName,
    ArrayList allowedUsers, int serviceKind, int commType, int bootType);
  public void AddServiceType(string typeName, string programName, int adminType,
    ArrayList allowedUsers, string description);
  public void Boot(string[] args);
  public void Connect(string userId, string password, CorsoStrategy strat, int
    aid, string serviceName, string corsoSite, string domain, int port);
  public CorsoNotification CreateAcceptor(CorsoStrategy strategy, string name,
    byte[] authorizationKey);
  public CorsoProcess CreateCompensateProcess(string entryName, CorsoData
    entryParam, CorsoTransaction tx, string serviceCreatorName, string
    serviceName, string siteName);
  public CorsoConstOid CreateConstOid(CorsoStrategy strategy);
  public CorsoConstOid[] CreateConstOids(CorsoStrategy strategy, int number);
  public CorsoData CreateData();
  public CorsoProcess CreateDependentProcess(string entryName, CorsoData
    entryParam, CorsoTransaction tx, string serviceCreatorName, string
    serviceName, string siteName);
  public CorsoProcess CreateIndependentProcess(string entryName, CorsoData
    entryParam, string serviceCreatorName, string serviceName, string siteName);
  public CorsoConstOid CreateNamedConstOid(CorsoStrategy strategy, string name,
    byte[] authorizationKey);
  public CorsoVarOid CreateNamedVarOid(CorsoStrategy strategy, string name,
    byte[] authorizationKey);
  public CorsoNotification CreateNotification(ArrayList notificationItems,
    CorsoStrategy strategy);
```

```
public CorsoProcess CreateOnAbortProcess(string entryName, CorsoData entryParam,
   CorsoTransaction tx, string serviceCreatorName, string serviceName, string
  siteName);
public CorsoProcess CreateOnCommitProcess(string entryName, CorsoData
  entryParam, CorsoTransaction tx, string serviceCreatorName, string
  serviceName, string siteName);
public CorsoSubTransaction CreateSubTransaction(CorsoTransaction tx);
public CorsoTopTransaction CreateTopTransaction();
public CorsoVarOid CreateVarOid(CorsoStrategy strategy);
public CorsoVarOid[] CreateVarOids(CorsoStrategy strategy, int number);
public void DeleteService(string serviceCreator, string serviceName);
public void DeleteServiceType(string typeCreator, string typeName);
public void DestroyLocalOids(CorsoOid[] oids, bool recursive);
public void DestroyOids(CorsoOid[] oids, bool recursive);
public void Disconnect();
public void DisconnectFromPartnerCorsoKernel(IPAddress inetAddress);
public override bool Equals(object anObject);
public void FreeOids(CorsoOid[] oids, bool recursive);
public CorsoNotification GetAcceptor(string name, byte[] authorizationKey);
public int GetCurrentAid();
public string GetCurrentCodePage();
public CorsoTransaction GetCurrentEntryTransaction();
public CorsoConstOid GetCurrentPid();
public string GetCurrentService();
public string GetCurrentServiceCreator();
public int GetCurrentServiceKind();
public int GetCurrentStartType();
public string GetCurrentStderr();
public string GetCurrentStdout();
public CorsoStrategy GetCurrentStrategy();
public string GetCurrentUser();
public bool GetCurrentWinConsoleFlag();
public string GetCurrentWinConsoleTitle();
public bool GetCurrentWinDetachedFlag();
public string GetCurrentWinDomain();
public string GetCurrentXDisplay();
public string GetCurrentXSetting();
public override int GetHashCode();
public DateTime GetLastMessageTimeStampOfCorsoKernel(IPAddress inetAddress);
public CorsoConstOid GetNamedConstOid(string name, string site, byte[]
  verificationKey, bool localCache, int timeout);
public CorsoVarOid GetNamedVarOid(string name, string site, byte[]
  verificationKey, bool localCache, int timeout);
public CorsoConstOid GetOrCreateNamedConstOid(CorsoStrategy strategy, string
  name, byte[] authorizationKey);
public CorsoVarOid GetOrCreateNamedVarOid(CorsoStrategy strategy, string name,
  byte[] authorizationKey);
public bool IsConnected();
public bool IsCorsoSuperUser();
public int ProcessEnd(int exitValue, int timeout);
public void SecureConnect(string userId, string password, CorsoStrategy strat,
  int aid, string serviceName, string corsoSite, string domain, int securePort);
public void SetCommType(string site, int commType);
public void SetCurrentCodePage(string newCodePage);
public void SetCurrentStartType(int newStartType);
public void SetCurrentStderr(string newStderr);
public void SetCurrentStdout(string newStdout);
public void SetCurrentStrategy(CorsoStrategy newStrategy);
public void SetCurrentUser(string newUser);
public void SetCurrentWinConsoleFlag(bool newWinConsoleFlag);
public void SetCurrentWinConsoleTitle(string newWinConsoleTitle);
public void SetCurrentWinDetachedFlag(bool newWinDetachedFlag);
public void SetCurrentWinDomain(string newWinDomain);
public void SetCurrentWinPassword(string newWinPassword);
public void SetCurrentXDisplay(string newXDisplay);
public void SetCurrentXSetting(string newXSetting);
public void ShutdownCorso();
public void UnregisterOidByName(CorsoTransaction tx, string name, byte[]
  verificationKey);
public void WaitForEntryTermination();

// Fields
public const int AUTO_START = 100;
public const int BOOT = 10;
```

```
  public const int COMM_TYPE_AUTO = 3;
  public const int COMM_TYPE_TCP = 2;
  public const int COMM_TYPE_UDP = 1;
  public const int CONNECT = 20;
  public const int DEPENDEND_PROCESS_TYPE = 1;
  public const int INDEPENDEND_PROCESS_TYPE = 2;
  public const int INFINITE_TIMEOUT = -1;
  public const int MANUAL_START = 200;
  public const int MAX_AID = 4;
  public const int NO_TIMEOUT = 0;
  public const int PERMANENT = 1;
  public const int PIPES = 1;
  public const int SHARED_MEMORY = 3;
  public const int SYSTEM_ADMINISTRATED = 2;
  public const int TCP_IP = 2;
  public const int TRANSIENT = 2;
  public const int USER_ADMINISTRATED = 1;
  public const int X_TRANSIENT = 4;
}
```

A.2 CorsoBase Class

```
public abstract class CorsoBase
{
  // Methods
  public CorsoConnection GetConnection();
}
```

A.3 CorsoStrategy Class

```
public class CorsoStrategy
{
  // Methods
  public CorsoStrategy(CorsoStrategy st);
  public CorsoStrategy(int aStrat);
  public override bool Equals(object stOther);
  public override int GetHashCode();

  // Fields
  public const int PR_DEEP = 1;
  public const int PR_DEEP_EAGER = 0x100001;
  public const int PR_DEEP_LAZY = 1;
  public const int PR_DEEP_READ_MAIN = 0x200001;
  public const int PR_DEEP_READ_NEXT = 1;
  public const int RELIABLE_0 = 0;
  public const int RELIABLE_1 = 0x800;
  public const int RELIABLE_2 = 0xc00;
}
```

A.4 CorsoShareable Interface

```
public interface CorsoShareable
{
  // Methods
  void Read(CorsoData data);
  void Write(CorsoData data);
}
```

A.5 CorsoOid Class

```
public abstract class CorsoOid : CorsoBase, CorsoShareable
{
  // Methods
  public int Aid();
  public int CompareTo(object obj);
  public void Destroy(bool recursive);
  public void DestroyLocal(bool recursive);
  public override bool Equals(object oidOther);
  public void Free(bool recursive);
  public ArrayList GetDistributionTopology();
  public override int GetHashCode();
  public string GetName();
  public CorsoStrategy GetStrategy();
  public bool IsZero();
  public int LastRequestNr();
  public void Name(string name, byte[] authorizationKey);
  public void Name(CorsoTransaction tx, string name, byte[] authorizationKey);
  public void Read(CorsoData data);
  public byte[] ReadBinary(CorsoTransaction tx, int timeout);
  public bool ReadBoolean(CorsoTransaction tx, int timeout);
  public void ReadData(CorsoData value, CorsoTransaction tx, int timeout);
  public double ReadDouble(CorsoTransaction tx, int timeout);
  public int ReadInt(CorsoTransaction tx, int timeout);
  public long ReadLong(CorsoTransaction tx, int timeout);
  public void ReadShareable(CorsoShareable value, CorsoTransaction tx, int
    timeout);
  public string ReadString(CorsoTransaction tx, int timeout);
  public override string ToString();
  public void Write(CorsoData data);
  public int WriteBinary(byte[] value, CorsoTransaction tx);
  public void WriteBinary(byte[] value, int timeout);
  public int WriteBinaryCompressed(byte[] value, CorsoTransaction tx);
  public void WriteBinaryCompressed(byte[] value, int timeout);
  public int WriteBoolean(bool value, CorsoTransaction tx);
  public void WriteBoolean(bool value, int timeout);
  public int WriteData(CorsoData value, CorsoTransaction tx);
  public void WriteData(CorsoData value, int timeout);
  public int WriteDouble(double value, CorsoTransaction tx);
  public void WriteDouble(double value, int timeout);
  public int WriteInt(int value, CorsoTransaction tx);
  public void WriteInt(int value, int timeout);
  public int WriteLong(long value, CorsoTransaction tx);
  public void WriteLong(long value, int timeout);
  public int WriteShareable(CorsoShareable value, CorsoTransaction tx);
  public void WriteShareable(CorsoShareable value, int timeout);
  public int WriteString(string value, CorsoTransaction tx);
  public void WriteString(string value, int timeout);
  public int WriteStringCompressed(string value, CorsoTransaction tx);
  public void WriteStringCompressed(string value, int timeout);

  // Fields
  public const int CONST = 2;
  public const int VAR = 4;
}
```

A.6 CorsoConstOid Class

```
public class CorsoConstOid : CorsoOid
{
  // Methods
  public CorsoConstOid();
  public CorsoConstOid(CorsoConstOid oid);
  public CorsoConstOid(CorsoConnection con, CorsoStrategy strategy);
  public bool Test(CorsoTransaction tx);
}
```

A.7 CorsoVarOid Class

```
public class CorsoVarOid : CorsoOid
{
  // Methods
  public CorsoVarOid();
  public CorsoVarOid(CorsoVarOid oid);
  public CorsoVarOid(CorsoConnection con, CorsoStrategy strategy);
  public int Append(CorsoTransaction tx, byte[] data);
  public byte[] Consume(CorsoTransaction tx, int timeout);
  public int GetTimeStamp();
  public void SetTimeStamp(int ts);
  public bool Test(CorsoTransaction tx, int timestamp);
  public override string ToString();
}
```

A.8 CorsoData Class

```
public class CorsoData : CorsoBase
{
  // Methods
  public byte[] GetBinary();
  public byte[] GetBinaryCompressed();
  public bool GetBoolean();
  public double GetDouble();
  public int GetInt();
  public void GetListTag();
  public long GetLong();
  public void GetShareable(CorsoShareable obj);
  public short GetShort();
  public string GetString();
  public string GetStringCompressed();
  public int GetStructTag(StringBuilder structName);
  public int PeekOidType();
  public byte PeekSignature();
  public int PeekStructTag(StringBuilder structName);
  public void PutBinary(byte[] val);
  public void PutBinaryCompressed(byte[] val);
  public void PutBoolean(bool val);
  public void PutDouble(double val);
  public void PutInt(int val);
  public void PutListTag();
  public void PutLong(long val);
  public void PutShareable(CorsoShareable obj);
  public void PutShort(short val);
  public void PutString(string val);
  public void PutStringCompressed(string val);
  public void PutStructTag(string structName, int arity);

  // Fields
  public const byte SIG_BINARY = 0x65;
  public const byte SIG_BINARY_COMPRESSED = 0x66;
  public const byte SIG_BOOLEAN = 0x6f;
  public const byte SIG_DOUBLE = 0x6d;
  public const byte SIG_INT = 0x6a;
  public const byte SIG_LIST = 110;
  public const byte SIG_LONG = 0x6b;
  public const byte SIG_OID = 100;
  public const byte SIG_STRING = 0x67;
  public const byte SIG_STRING_COMPRESSED = 0x68;
  public const byte SIG_STRUCT = 0x69;
}
```

A.9 CorsoTransaction Class

```
public abstract class CorsoTransaction : CorsoBase
{
  // Methods
  public void Abort();
  public void CancelRequest(int reqNr);
  public override bool Equals(object txOther);
  public override int GetHashCode();
  public override string ToString();
  public CorsoTransFailInfo TransFailInfo();
}
```

A.10 CorsoTopTransaction Class

```
public class CorsoTopTransaction : CorsoTransaction
{
  // Methods
  public CorsoTopTransaction();
  public CorsoTopTransaction(CorsoConnection con);
  public void CCommit(int timeout);
  public void Commit(int timeout);
  public void Create(CorsoConnection con);
  public void TryCCommit(int timeout);
  public void TryCommit(int timeout);
}
```

A.11 CorsoNotification Class

```
public class CorsoNotification : CorsoBase, CorsoShareable
{
  // Methods
  public CorsoNotification();
  public CorsoNotification(CorsoVarOid notifOid);
  public void AddItem(CorsoNotificationItem item, CorsoTransaction tx);
  public void AddItem(CorsoNotificationItem item, int timeout);
  public void AddItems(ArrayList items, CorsoTransaction tx);
  public void AddItems(ArrayList items, int timeout);
  public void ConfigureItem(CorsoNotificationItem item, int timeout);
  public void ConfigureItems(ArrayList items, int timeout);
  public override bool Equals(object notifOther);
  public override int GetHashCode();
  public ArrayList GetInvalidItems();
  public ArrayList GetItems();
  public ArrayList GetItemsForCurrentConnection();
  public CorsoVarOid NotificationId();
  public void Pause(CorsoProcess process);
  public void Read(CorsoData data);
  public void RemoveItem(CorsoNotificationItem item, CorsoTransaction tx);
  public void RemoveItem(CorsoNotificationItem item, int timeout);
  public void RemoveItems(ArrayList items, CorsoTransaction tx);
  public void RemoveItems(ArrayList items, int timeout);
  public void Reset();
  public CorsoNotificationItem Start(int timeout, CorsoData data);
  public override string ToString();
  public void Write(CorsoData data);
}
```

A.12 CorsoNotificationItem Class

```
public class CorsoNotificationItem : CorsoBase
{
  // Methods
  public CorsoNotificationItem(CorsoNotificationItem r);
  public CorsoNotificationItem(CorsoConstOid oid, int flags);
  public CorsoNotificationItem(CorsoVarOid oid, int mytimeStamp, int flags);
  public int Configuration();
  public void Configuration(int flags);
  public CorsoConstOid ConstOid();
  public void ConstOid(CorsoConstOid oid);
  public override bool Equals(object anObject);
  public override int GetHashCode();
  public int ObjectType();
  public int TimeStamp();
  public void TimeStamp(int ts);
  public override string ToString();
  public CorsoVarOid VarOid();
  public void VarOid(CorsoVarOid oid);

  // Fields
  public const int CURRENT_TIMESTAMP = 0x10;
  public const int IGNORE_ITEM = 1;
  public const int INCREMENT_TIMESTAMP = 8;
  public const int INITIALIZE_WITH_CURRENT_TIMESTAMP = -2;
  public const int PAUSE_NOTIF = 4;
  public const int RESET_NOTIF = 2;
}
```

A.13 CorsoException Class

```
public class CorsoException : Exception
{
  // Methods
  public CorsoException(int p);
  public CorsoException(int p, int s);
  public string GetMessage();
  public int Primary();
  public int Secondary();
  public override string ToString();
  public int UserError();

  // Fields
  public const int DATA = -101;
  public const int DATA_TAG = -102;
  public const int ERR_ABORT_TX = -11;
  public const int ERR_ADD_NOTIFICATION_ITEMS = -51;
  public const int ERR_ADD_SERVICE = -42;
  public const int ERR_ADD_SERVICE_TYPE = -40;
  public const int ERR_ALREADY_CONNECTED = -32;
  public const int ERR_C_COMMIT_TX = -13;
  public const int ERR_CANCEL_REQ = -18;
  public const int ERR_COMMIT_TX = -12;
  public const int ERR_COMPENSATE_PROCESS = -7;
  public const int ERR_CONFIG_NOTIFICATION_ITEMS = -56;
  public const int ERR_CONNECTING_FAILED = -30;
  public const int ERR_CONNECTION_BROKEN = -33;
  public const int ERR_CONSUME = -3;
  public const int ERR_CREATE_ACCEPTOR = -84;
  public const int ERR_CREATE_NAMED_OBJECT = -60;
  public const int ERR_CREATE_NOTIFICATION = -50;
  public const int ERR_CREATE_OBJECT = -1;
  public const int ERR_CREATE_TX = -10;
  public const int ERR_DELETE_NOTIFICATION_ITEMS = -52;
  public const int ERR_DELETE_SERVICE = -43;
  public const int ERR_DELETE_SERVICE_TYPE = -41;
  public const int ERR_DEP_PROCESS = -5;
```

```
public const int ERR_DISCONNECT_FROM_PARTNER_SITE = -82;
public const int ERR_EXPORT_OBJECT = -23;
public const int ERR_FREE_OBJECT = -24;
public const int ERR_GET_ACCEPTOR = -85;
public const int ERR_GET_DISTRIBUTION_TOPOLOGY = -28;
public const int ERR_GET_NAME = -27;
public const int ERR_GET_NAMED_OBJECT = -61;
public const int ERR_GET_STRATEGY = -26;
public const int ERR_INDEP_PROCESS = -6;
public const int ERR_INTERRUPTED = -95;
public const int ERR_INVALID_PARAMETER = -89;
public const int ERR_LAST_MSG_FROM_CORSO = -81;
public const int ERR_NO_CORSO_CONNECTION = -80;
public const int ERR_NOT_CONNECTED = -31;
public const int ERR_NOTIFICATION_STOPPED = -59;
public const int ERR_OBJECT_DESTROY = -25;
public const int ERR_OK = 0;
public const int ERR_ON_ABORT_PROCESS = -9;
public const int ERR_ON_COMMIT_PROCESS = -8;
public const int ERR_PAUSE_NOTIFICATION = -55;
public const int ERR_PREPARE_TX = -14;
public const int ERR_PROCESS_END = -20;
public const int ERR_PROCESS_SIGNAL = -21;
public const int ERR_PROCESS_STATE = -22;
public const int ERR_QUERY_INVALID_NOTIFICATION_ITEMS = -58;
public const int ERR_QUERY_NOTIFICATION_ITEMS = -57;
public const int ERR_READ = -2;
public const int ERR_REGISTER_NAMED_OBJECT = -62;
public const int ERR_RESET_NOTIFICATION = -54;
public const int ERR_SET_COMM_TYPE = -83;
public const int ERR_SHUTDOWN = -71;
public const int ERR_START_NOTIFICATION = -53;
public const int ERR_STRATEGY = -44;
public const int ERR_TEST_CONST = -72;
public const int ERR_TEST_VAR = -73;
public const int ERR_TRANSACTION_FAIL_INFO = -45;
public const int ERR_TRY_C_COMMIT_TX = -16;
public const int ERR_TRY_COMMIT_TX = -15;
public const int ERR_TRY_PREPARE_TX = -17;
public const int ERR_UNREGISTER_NAMED_OBJECT = -63;
public const int ERR_USER = -19;
public const int ERR_USER_AID_INCOMPATIBILITY = -14;
public const int ERR_USER_AID_NOT_SUPPORTET = -27;
public const int ERR_USER_CANT_SEND_SIGNAL = -15;
public const int ERR_USER_CANT_START_COMPENSATE_ACTION = -16;
public const int ERR_USER_CANT_START_ON_COMMIT_ACTION = -17;
public const int ERR_USER_DUPLICATE_NOTIFY_OBJECT = -69;
public const int ERR_USER_DUPLICATE_WRITE_REQUEST = -24;
public const int ERR_USER_EMPTY_VALUE = -25;
public const int ERR_USER_FOREIGN_PID = -56;
public const int ERR_USER_ILLEGAL_BOOT_TYPE = -29;
public const int ERR_USER_ILLEGAL_CANCEL_REQUEST_NUMBER = -30;
public const int ERR_USER_ILLEGAL_COMMIT_TYPE = -31;
public const int ERR_USER_ILLEGAL_DATA = -34;
public const int ERR_USER_ILLEGAL_ENTRY_DATA = -32;
public const int ERR_USER_ILLEGAL_EXIT_VALUE_OR_SIGNAL = -33;
public const int ERR_USER_ILLEGAL_LOGICAL_TIME_STAMP = -43;
public const int ERR_USER_ILLEGAL_NAME = -204;
public const int ERR_USER_ILLEGAL_NOTIFICATION_CONFIG = -70;
public const int ERR_USER_ILLEGAL_OID = -37;
public const int ERR_USER_ILLEGAL_PID = -38;
public const int ERR_USER_ILLEGAL_PROCESS_START_TYPE = -39;
public const int ERR_USER_ILLEGAL_REQUESTING_PROCESS = -35;
public const int ERR_USER_ILLEGAL_SERVICE = -10;
public const int ERR_USER_ILLEGAL_STRATEGY = -41;
public const int ERR_USER_ILLEGAL_TRANSACTION = -42;
public const int ERR_USER_INVALID_TRANSACTION = -22;
public const int ERR_USER_MAXIMUM_OID_NUMBER_EXCEEDED = -21;
public const int ERR_USER_NAME_ALREADY_USED = -202;
public const int ERR_USER_NAMED_OBJECT_NOT_FOUND = -205;
public const int ERR_USER_NOT_AUTHORIZED_TO_CONNECT = -12;
public const int ERR_USER_NOT_NOTIFICATION = -71;
public const int ERR_USER_NOTIFICATION_NOT_RUNNING = -74;
public const int ERR_USER_NOTIFICATION_RUNNING = -73;
```

```
  public const int ERR_USER_OBJECT_ALREADY_NAMED = -201;
  public const int ERR_USER_OBJECT_CREATION_ERROR = -18;
  public const int ERR_USER_OBJECT_LOST = -47;
  public const int ERR_USER_OBJECT_NOT_IN_NOTIFICATION = -75;
  public const int ERR_USER_PID_NOT_AVAILABLE = -48;
  public const int ERR_USER_PREPARE_NAME_REQUEST = -80;
  public const int ERR_USER_PREPARE_TOP_TRANSACTION = -49;
  public const int ERR_USER_PROCESS_ALREADY_EXITING = -51;
  public const int ERR_USER_PROCESS_CREATION_ERROR = -19;
  public const int ERR_USER_PROCESS_STATE = -54;
  public const int ERR_USER_PROCESS_TERMINATED = -55;
  public const int ERR_USER_READ_TEST_NOTIFICATION = -68;
  public const int ERR_USER_SIGNAL_REMOTE_PROCESS = -52;
  public const int ERR_USER_STRATEGY_INCONSISTENCY = -53;
  public const int ERR_USER_SUPERUSER = -81;
  public const int ERR_USER_TRANSACTION_CREATION_ERROR = -20;
  public const int ERR_USER_TRANSACTION_IS_PREPARED = -50;
  public const int ERR_USER_TRANSACTION_NOT_RUNNING = -57;
  public const int ERR_USER_TX_IS_ALREADY_ABORTED = -13;
  public const int ERR_USER_UNAUTHORIZED_OBJECT_ACCESS = -58;
  public const int ERR_USER_UNKNOWN_INTERNET_ADDRESS = -59;
  public const int ERR_USER_UNSUPPORTED_DATA_TRANSLATION = -26;
  public const int ERR_USER_USER_NOT_TRUSTED = -11;
  public const int ERR_USER_WRITE_NOTIFICATION = -67;
  public const int ERR_USER_WRITE_PID = -60;
  public const int ERR_USER_WRONG_ARGUMENT = -28;
  public const int ERR_USER_WRONG_DATA = -61;
  public const int ERR_USER_WRONG_PROCESS_DEPENDENCY_TYPE = -62;
  public const int ERR_USER_ZERO_OID = -64;
  public const int ERR_USER_ZERO_PID = -63;
  public const int ERR_WRITE = -4;
  public const int ERR_WRONG_BOOT_PARAMETER = -70;
  public const int TIMEOUT = -100;
}
```

A.14 CorsoDataException Class

```
public class CorsoDataException : CorsoException
{
  // Methods
  public CorsoDataException();
  public override string ToString();
}
```

A.15 CorsoReadException Class

```
public class CorsoReadException : CorsoException
{
  // Methods
  public CorsoReadException();
}
```

A.16 CorsoWriteException Class

```
public class CorsoWriteException : CorsoException
{
  // Methods
  public CorsoWriteException();
}
```

A.17 CorsoTimeoutException Class

```
public class CorsoTimeoutException : CorsoException
{
  // Methods
  public CorsoTimeoutException(int msg, bool convertFromMsgTag);
  public int MessageCode();
  public override string ToString();

  // Fields
  public const int C_COMMIT_TX = -9;
  public const int COMMIT_TX = -7;
  public const int CONSUME = -17;
  public const int GET_NAMED_CONST_OID = -16;
  public const int GET_NAMED_VAR_OID = -15;
  public const int NOTIFICATION_ADD = -4;
  public const int NOTIFICATION_REMOVE = -5;
  public const int NOTIFICATION_START = -6;
  public const int PREPARE_TX = -11;
  public const int PROCESS_END = -3;
  public const int READ = -1;
  public const int TEST_CONST = -13;
  public const int TEST_VAR = -14;
  public const int TRY_C_COMMIT_TX = -10;
  public const int TRY_COMMIT_TX = -8;
  public const int TRY_PREPARE_TX = -12;
  public const int WRITE = -2;
}
```

A.18 CorsoTransactionException Class

```
public class CorsoTransactionException : CorsoException
{
  // Methods
  public CorsoTransactionException();
}
```

A.19 CorsoTransFailInfo Class

```
public class CorsoTransFailInfo
{
  // Methods
  public int FailureType();
  public string FailureTypeToString();
  public int RequestCommandTag();
  public string RequestCommandTagToString();
  public CorsoConstOid RequestConstOid();
  public int RequestNr();
  public int RequestOidType();
  public CorsoConstOid RequestPid();
  public CorsoTransaction RequestTx();
  public CorsoVarOid RequestVarOid();
  public override string ToString();

  // Fields
  public const int DEPENDENT_SUBPROCESS_ABORTED = -4;
  public const int DUPLICATE_WRITE = -15;
  public const int DUPLICATE_WRITE_IN_SUBTRANSACTION = -11;
  public const int ILLEGAL_TRANSACTION_STATE = -14;
  public const int MAIN_COPY_OF_OBJECT_NOT_AVAILABLE = -17;
  public const int NO_DIAGNOSIS = -1;
  public const int OBJECT_LOST = -16;
  public const int OBJECT_REQUEST_FAILED = -7;
```

```
  public const int OPEN_SUBTRANSACTION = -18;
  public const int READ_VAR_FAILED = 5;
  public const int READ_VAR_WITHOUT_TIMEOUT_FAILED = 6;
  public const int REGISTER_FAILED = -20;
  public const int REQUEST_NOT_AVAILABLE = 0;
  public const int STRATEGY_INCONSISTENCY = -8;
  public const int SUBTRANSACTION_ABORTED = -2;
  public const int SUBTRANSACTION_START_FAILED = 7;
  public const int TEST_CONST_FAILED = 1;
  public const int TEST_VAR_FAILED = 2;
  public const int TRANS_TIMEOUT = -19;
  public const int TTR_STRATEGY_INCONSISTENCY = -9;
  public const int UNREGISTER_FAILED = -21;
  public const int WRITE_CONST_FAILED = 3;
  public const int WRITE_VAR_FAILED = 4;
}
```

Appendix B

SpaceMap Data Structure

In the course of XVSM's development and conception, a complete distributed hash-based key-value container (a hash map) was created in $C^{\#}$ as a prototype for XVSM's advanced core-supported coordination types. This so-called *SpaceMap* was implemented on top of the CORSO space-based middleware and its .NET &Co language binding.

Figure B.1 shows a diagram illustrating the structure of the SpaceMap implementation and its most important classes in a simplified form. Programmers use instances of the SpaceMap via the static factory class *SpaceMapFactory*, which allows maps to be newly created, looked up, and looked up or created (cf. the *SpaceObjectFactory* class from chapter 5). The factory provides the caller with an instance of the generic *SpaceMap<TKey, TValue>* class, which constitutes the public interface of SpaceMap.

The *SpaceMap<>* class provides typical collection-like methods for adding new items, checking whether the map contains a certain item, getting the value associated with a key, etc. In order to be synchronized with the space (in a space transaction), the class supports methods for refreshing and persisting the SpaceMap from and to the space, and it also has two notification events that get fired when the SpaceMap or its elements change, no matter whether the change occurred locally or somewhere else on the network.

Internally, *SpaceMap<>* indirectly delegates to a low-level class called *HashtableStructure<>*, which represents the shareable data dictionary actually stored within the space, providing space serialization methods and functionality operating on one local deserialized instance of the SpaceMap. It is is wrapped by a *HashtableStructureWrapper<>* class, which implements higher-level functionality the actual *SpaceMap<>* class can build upon, such as finding the key or value space objects for given key data. In addition, the wrapper holds the CORSO OID object identifying the SpaceMap object in the space and is configured with the replication strategies to be used for map key and value data. By wrapping the map manipulation methods of the structure class with calls to *Refresh* and *Persist*, it implements higher-level operations and causes the structure to always be synchronized with the space.

To implement the notification events, the wrapper holds a reference to a *NotificationListener* class (cf. chapter 5), which handles threading and CORSO notification management.

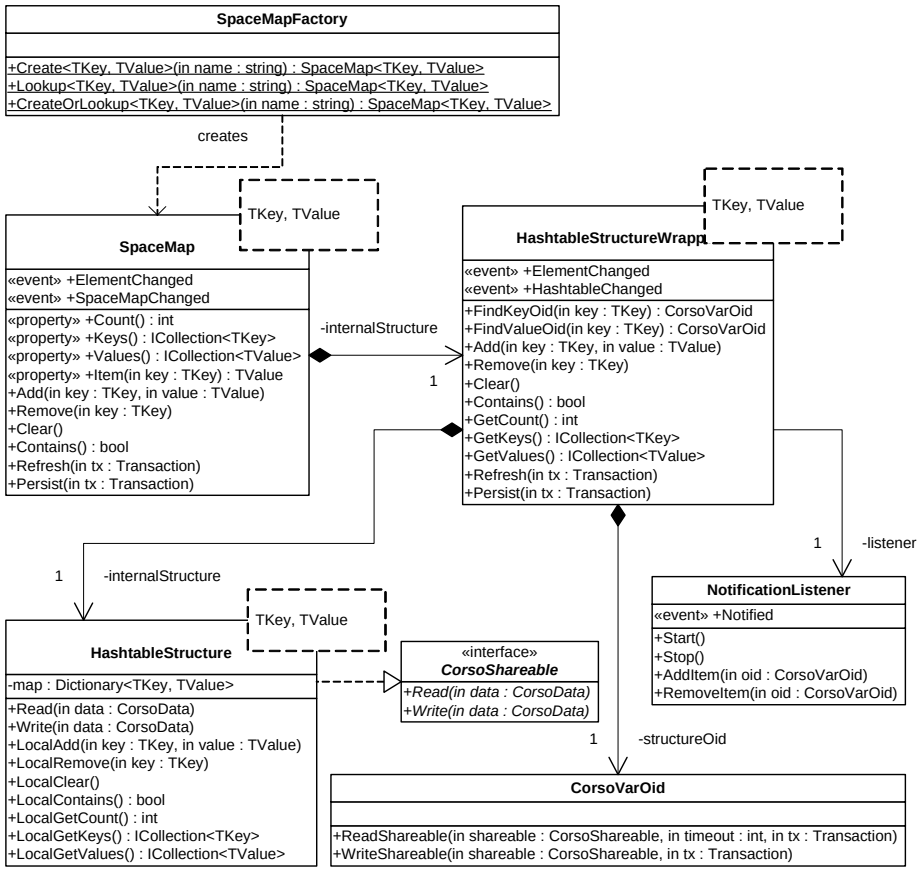

Figure B.1: SpaceMap class diagram

The following section contains the public API of the SpaceMap, automatically generated with the free *Reflector* .NET tool. The SpaceMap's full source code is too long to be included in this thesis, but it can be obtained from the author on request.

```
public class SpaceMap<TKey, TValue> : IDictionary<TKey, TValue>,
  ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey,
  TValue>>, IEnumerable
{
      // Events
      public event ElementChangedEventHandler<TKey, TValue> ElementChanged;
      public event SpaceMapChangedEventHandler<TKey, TValue> SpaceMapChanged;

      // Methods
      public SpaceMap(Connection connection, CorsoVarOid mapOid);
      public void Add(KeyValuePair<TKey, TValue> item);
      public void Add(KeyValuePair<TKey, TValue> item, Transaction tx);
      public void Add(TKey key, TValue value);
      public void Add(TKey key, TValue value, Transaction tx);
      public bool AddAndWait(TKey key, out TValue value, TimeSpan timeout);
```

```
public void AddLink<TKey1, TValue1>(string name, SpaceMap<TKey1, TValue1>
  linkedMap);
public IEnumerable<KeyValuePair<TKey, TValue>> AsEnumerable(Transaction tx);
public void Clear();
public void Clear(Transaction tx);
public bool Contains(KeyValuePair<TKey, TValue> item);
public bool Contains(KeyValuePair<TKey, TValue> item, Transaction tx);
public bool ContainsKey(TKey key);
public bool ContainsKey(TKey key, Transaction tx);
public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex);
public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex,
  Transaction tx);
public void Delete();
public void ForAll(ElementAction<TKey, TValue> action, Transaction tx);
public void ForAllRead(ElementAction<TKey, TValue> readAction);
public void ForAllWrite(ElementAction<TKey, TValue> action);
public bool GetAndWait(TKey key, out TValue value, TimeSpan timeout);
public int GetCount(Transaction tx);
public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();
public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator(Transaction
  tx);
public ICollection<TKey> GetKeys(Transaction tx);
public SpaceMap<TKey1, TValue1> GetLink<TKey1, TValue1>(string name);
public ICollection<TValue> GetValues(Transaction tx);
private void ImmediateRefresh(CorsoData data);
private void internalStructure_ElementChanged(CorsoVarOid valueOid,
  CorsoData valueData);
private void internalStructure_HashtableChanged();
private void OnElementChanged(ElementChangedEventArgs<TKey, TValue> args);
private void OnSpaceMapChanged();
public void Persist(Transaction tx);
public void Refresh();
public void Refresh(Transaction tx);
public bool Remove(KeyValuePair<TKey, TValue> item);
public bool Remove(TKey key);
public bool Remove(TKey key, Transaction tx);
public bool Remove(KeyValuePair<TKey, TValue> item, Transaction tx);
IEnumerator IEnumerable.GetEnumerator();
public bool TryGetValue(TKey key, out TValue value);
public bool TryGetValue(TKey key, out TValue value, Transaction tx);

// Properties
public Connection Connection { get; }
public int Count { get; }
public bool IsReadOnly { get; }
public TValue this[TKey key, Transaction tx] { get; set; }
public TValue this[TKey key] { get; set; }
public ICollection<TKey> Keys { get; }
public IEnumerable<string> Links { get; }
public ISynchronizeInvoke SynchronizeInvoke { get; set; }
public ICollection<TValue> Values { get; }

// Fields
private ElementChangedEventHandler<TKey, TValue> ElementChanged;
private HashtableStructureWrapper<TKey, TValue> internalStructure;
private object monitor;
private SpaceMapChangedEventHandler<TKey, TValue> SpaceMapChanged;
private ISynchronizeInvoke synchronizeInvoke;

// Nested Types
public delegate void ElementAction(TKey key, TValue value);

public class ElementChangedEventArgs
{
      // Methods
      internal ElementChangedEventArgs(SpaceMap<TKey, TValue> map,
        CorsoVarOid valueOid, CorsoData valueData);
      public TKey GetKey();
      public TKey GetKey(Transaction tx);
      private CorsoVarOid GetKeyOid(Transaction tx);
      public TValue GetValue();
      public TValue GetValue(Transaction tx);

      // Fields
```

```
            private CorsoVarOid keyOid;
            public readonly SpaceMap<TKey, TValue> Map;
            private CorsoData valueData;
            private CorsoVarOid valueOid;
        }

        public delegate void ElementChangedEventHandler(SpaceMap<TKey, TValue>
          sender, SpaceMap<TKey, TValue>.ElementChangedEventArgs args);
        public delegate void SpaceMapChangedEventHandler(SpaceMap<TKey, TValue>
          map);
}
```

Listing B.1: SpaceMap's public interface

Glossary

.NET Base Class Library — the class library coming with the .NET platform, 27

.NET platform — an environment for object-oriented computer applications which are executed ("managed", as opposed to "unmanaged" or "native" applications) by an execution environment as defined by the Common Language Infrastructure specification, 27

adoptability — the property of a tool being easy to integrate into an existing system or development process, 58

advice — a piece of code executed by an aspect at a specific set of join points, 34

annotation — a descriptive item declaratively attached to a program element, 27

AO-DCL — *Aspect-Oriented Distributed Concern Library*, a library of declaratively applicable, cleanly encapsulated cross-cutting concerns of space-based computing, 111

AOP — *Aspect-Oriented Programming*, a new software development paradigm allowing cross-cutting concerns to be cleanly encapsulated, 31

AOP infrastructure — a platform, toolkit, or framework providing a weaving mechanism for building an AOP tool, 37

API — *application programming interface*, 22

aspect — an entity of modularization for a cross-cutting concern, similar as a class is an entity of modularization for a functional concern, 32

aspect language — the feature set of an aspect-oriented tool and the programming mechanisms provided to make use of this feature set, 60

AspectJ — a very mature AOP tool for the Java platform, 33

augmenting weaving — the act of combining aspects and classes by extending their in-memory form via reflective access, 37

C# an object-oriented programming language created for the .NET platform, 27

CLI *Common Language Infrastructure*, the specification defining the .NET platform, 27

CLR *Common Language Runtime*, an implementation of the Common Language Infrastructure by the Microsoft Corporation, 27

concern a software requirement, 25

CORSO *Coordinated Shared Objects*, a space-based middleware extending the original tuple spaces concept, 17

CORSO constant object a CORSO data object that can be written only once, 18

CORSO data buffer a buffer receiving serialized data when an object is written to the space and holding deserialized data when an object is read from the space, 23

CORSO shareable object an object which can be serialized to the CORSO space because it implements the *CorsoShareable* interface, 23

CORSO strategy the combination of reliability and replication flags for a CORSO data object, 23

CORSO variable object a CORSO data object that can be written multiple times, 18

cross-cutting concern a requirement to an application whose implementation cannot be cleanly encapsulated by traditional (i.e. procedural or object-oriented) modularization mechanism but instead couples several otherwise unrelated modules, 25

custom attribute a descriptive item declaratively attached to a program element such as a class, method, field, etc., 27

custom attribute class the class defining a new kind of custom attribute, 28

custom attributes an implementation of extensible metadata on the .NET platform, 27

declarative programming a form of programming where the developer describes an application's goals rather than specifying the exact algorithms to reach them, 5

deserialization the process of restoring an object's state from a persistable form, 23

dynamic weaving another name for runtime weaving, 38

eager replication a flag in the PR_{deep} protocol indicating that changes committed to a space object should be replicated to other processes as fast as possible, 21

extensibility the property of a tool allowing new concerns to be integrated even though they were not considered at the tool's design time, 57

extensible metadata a concept allowing to attach programmer-defined descriptive information to the source code elements of a program, 27

factory a class, object, or method used for creating (and usually configuring) new object instances, 58

IL *(Common) Intermediate Language*, a standardized (virtual) machine language which computer programs targeting the .NET platform are compiled to, 37

imperative programming a style of programming where the developer gives step-by-step instructions, implementing algorithms to solve the application's goals, 5

inter-type declaration another name for the introduction concept, 35

interception the act of stopping a program's execution at a certain point, allowing specific *interception handling code* to be executed and to decide how and whether to go on with executing the intercepted code, 36

intermediate code weaving the act of combining aspects and classes in an intermediate code form, 37

introducer a nested class or field of an aspect implementing an interface which is added to the static structure of a target class by means of introduction, 82

introduction the mechanism allowing an aspect to change the static structure of a class, 35

JavaSpaces a recent implementation of the Linda concept, 16

join point a point during the dynamic execution of an application where an aspect can influence the program, 32

join point model the set of join point kinds supported by an AOP tool, 38

language binding the API necessary to access a middleware from a given programming language, 22

lazy replication a flag in the PR_{deep} protocol indicating that changes committed to a space object should be replicated to other processes only on demand, 21

LCG *Lightweight Code Generation*, a mechanism for generating methods at runtime and attaching them to any existing type, 49

light weight the property of a tool having minimal impact on a system, 57

Linda the original formulation and implementation of tuple spaces, 13

load-time weaving the act of combining aspects and classes just as they are loaded into memory, 38

notification a mechanism for informing processes about changes conducted to a space object in near real-time, 22

object-oriented programming a way of imperative programming where the developer encapsulates related data (fields) and functionality (methods) into distinct modules (objects/classes), providing mechanisms such as object referencing, subclassing, method overriding, and polymorphism for building large applications, 6

obliviousness the property of a programmer not being (or not needing to be) aware of what cross-cutting concerns affect a class, or the property of a class not being specifically designed to allow for cross-cutting concerns to affect it, 35

OID *object identifier*, an opaque value referencing a data object in the CORSO space, 18

orthogonal concern a requirement that has no side-effects (in concept) on the functional requirements of an application; in implementation, these often cross-cut object-oriented designs, 25

pointcut a mechanism for selecting a subset of the join points existing in an application, 34

PR_{deep} the replication protocol used by CORSO, 20

primary copy the main replica of a CORSO object in the PR_{deep} protocol, 20

primary copy migration the act of transferring the status of *primary copy* from one replica of a CORSO object to another one, necessary for CORSO's transaction handling mechanism, 20

quantification the possibility of declaratively selecting a set of execution points during an application's execution without needing to imperatively execute code at these points, 35

reflection a mechanism for inspecting the internals of any object at runtime, without the object providing explicit support for this, 83

reliable0, reliable1, reliable2 the different persistence levels supported by the CORSO middleware, 22

runtime weaving the act of combining aspects and classes after they have been loaded into memory, while the application is executing, 38

serialization the process of extracting an object's state into a persistable form, 23

source code annotations an implementation of extensible metadata on the Java platform, 29

source code weaving the act of combining aspects and classes in source code form, 37

space a virtual memory shared by different processes for communication and coordination purposes, 13

SpaceMap a distributed key-value container for the space, 221

static weaving the act of combining aspects and classes before they are loaded into memory, 38

subclass proxy an AOP infrastructure mechanism where an instance of a class is substituted by an instance of a subclass which contains code allowing aspects to affect the class, 43

target code the code affected by an aspect, 33

target object an object affected by an aspect, 33

transaction a mechanism for combining multiple operations into a single atomic and isolated action with consistent and durable results, 19

tuple a finite, structured sequence of heterogeneous data, 13

tuple space a tuple store shared by different processes for communication and coordination purposes, 13

weaving the process of combining aspects and classes into one executable program, 32

XL-AOF *Extensible Lightweight Aspect-Oriented Framework*, an aspect-oriented toolkit specifically aimed at space-based computing and adoptability, 57

XVSM *Extensible Virtual Shared Memory*, a new, sophisticated space-based network abstraction layer, successor to CORSO, 24

XVSM container a referenceable data object in the XVSM space holding data values, 24

Bibliography

[AB+03] Rey Abe, Martin Beinhart, et al. Need for rigorous methods and tools in collaborative mobile software solutions – a use case study on a travel service application. Technical report, Vienna University of Technology, E185/1, Vienna, Austria, 2003.

[And81] Gregory R. Andrews. Synchronizing resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, 1981.

[B+04] Gilad Bracha et al. Jsr 175: A metadata facility for the java™ programming language, final release. Technical report, Sun Microsystems, 2004. Available from: `http://jcp.org/en/jsr/detail?id=175`.

[Bey01] Derek Beyer. *C# Com+ Programming*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[BHM02] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 42–122, London, UK, 2002. Springer-Verlag.

[BM02] Julia Case Bradley and A. C. Millspaugh. *Programming in Visual Basic.Net*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[Bon04] Jonas Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[BP01] Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object serialization. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 173–180, New York, NY, USA, 2001. ACM Press.

[Bur04] Bill Burke. *JBoss Aspect-Oriented Programming (AOP)*, 2004. Available from: `http://www.jboss.org/products/aop`.

[CD00] George F. Coulouris and Jean Dollimore. *Distributed systems: Concepts and Design (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988 and 2000.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CG01] Nicholas Carriero and David Gelernter. A computational model of everything. *Communications of the ACM*, 44(11):77–81, 2001.

[CS06] Alan Cyment and Fabian Schmied. An analysis on existing and potential weaving mechanisms for the .net framework. In *Proceedings Second International Conference on Innovative Views of .NET Technologies IVNET'06*, Florianópols, Brazil, 2006.

[D+06] Linda DeMichiel et al. Jsr 220: Enterprise javabeans™ 3.0. Technical report, Sun Microsystems, 2006. Available from: `http://jcp.org/en/jsr/detail?id=220`.

[Dei05] Matthew Deiters. Aspect-oriented programming. Technical report, Microsoft Corporation, 2005. Available from: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/AOPArticle.asp`.

[ECM05a] ECMA International. Standard ECMA-334 – C# language specification, 3rd edition. Technical report, ECMA International, 2005. Available from: `http://www.ecma-international.org/publications/standards/ecma-334.htm`.

[ECM05b] ECMA International. Standard ECMA-335 – common language infrastructure (CLI), 3rd edition. Technical report, Ecma International, 2005. Available from: `http://www.ecma-international.org/publications/standards/ecma-335.htm`.

[ECM05c] ECMA International. Standard ECMA-372 – c++/cli language specification. Technical report, Ecma International, 2005. Available from: `http://www.ecma-international.org/publications/standards/Ecma-372.htm`.

[eK94] eva Kühn. Fault-tolerance for communicating multidatabase transactions. In *IEEE Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS, Wailea, Maui, Hawaii, January 4–7 1994)*, pages 323–332, 1994.

[eK96] eva Kühn. A distributed and recoverable linda implementation with prolog &co. In *Proceedings of the Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)*, Miskolc, Hungary, 1996.

[eK01] eva Kühn. Preface: How to approach the virtual shared memory paradigm. *Virtual shared memory for distributed architectures*, pages .7–.22, 2001.

[eKBM05] eva Kühn, Martin Beinhart, and Martin Murth. Improving data quality of mobile internet applications with an extensible virtual shared memory approach. In *Proceedings WWW/Internet 2005 (Lisbon, Portugal, October 19–22 2005)*, 2005.

[eKFS06] eva Kühn, Gerald Fessl, and Fabian Schmied. Aspect-oriented programming with runtime-generated subclass proxies and .net dynamic methods. *Journal of .NET Technologies*, 4:17–24, 2006.

[eKN98] eva Kühn and Georg Nozicka. Post-client/server coordination tools. *Coordination Technology for Collaborative Applications, Springer Series Lecture Notes in Computer Science, Vol. 1364*, pages 231–253, 1998.

[eKPE92] eva Kühn, Franz Puntigam, and Ahmed K. Elmagarmid. An execution model for distributed database transactions and its implementation in vpl. In *EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*, pages 483–498, London, UK, 1992. Springer-Verlag.

[eKRJ05] eva Kühn, Johannes Riemer, and Geri Joskovicz. XVSM (extensible virtual shared memory) architecture and application. Technical report, Space-Based Computing Group, Institute of Computer Languages, Vienna University of Technology, 2005.

[eKS05a] eva Kühn and Fabian Schmied. Attributes &Co – collaborative applications with declarative shared objects. In *Proceedings WWW/Internet 2005, Lisbon, Portugal, October 19–22 2005*, pages 427–434, 2005.

[eKS05b] eva Kühn and Fabian Schmied. XL-AOF – lightweight aspects for space-based computing. In *Proceedings Workshop on Aspect-Oriented Middleware Development (AOMD, Grenoble, France, November 28, 2005), Article No. 2*, 2005.

[ELLR90] Ahmed K. Elmagarmid, Yungho Leu, Witold Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–518, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, 2000.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, Greenwich, CT, USA, 2004.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Professional, Boston, USA, 1999.

[For95] Alexander Forst. Implementation of the coordination language c&co. Master's thesis, Vienna University of Technology, Vienna, Austria, 1995.

[FPB87] Jr. Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer Magazine (first published in Information Processing 1986)*, 20(4):10–19, 1987.

[FR03] Marc Fleury and Francisco Reverbel. The jboss extensible server. In *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference, Lecture Notes in Computer Science*, volume 2672, pages 344–373, 2003.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, USA, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, Boston, USA, 2005.

[Gro04] The Standish Group. 2004 third quarter research report. Technical report, The Standish Group, 2004. Available from: `http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf`.

[GS+06] William G. Griswold, Kevin Sullivan, et al. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[Har05] Allan M. Hart. Hibernate in the classroom. *Journal of Computing Sciences in Colleges*, 20(4):98–100, 2005.

[Hir03] Robert Hirschfeld. AspectS - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[JH+05] Rod Johnson, Juergen Hoeller, et al. *Spring - Java/J2EE Application Framework*, 2005. Available from: `http://static.springframework.org/spring/docs/1.2.x/spring-reference.pdf`.

[Joh05] Roger Johansson. *NAspect AOP engine*, 2005. Available from: `http://blogs.wdevs.com/phirephly/archive/2005/11/23/11308.aspx`.

[JP+06] Rod Johnson, Mark Pollack, et al. *Spring .NET Reference Documentation*, 2006. Available from: `http://www.springframework.net/doc/reference/pdf/spring-net-reference.pdf`.

[Ker02] Mik Kersten. Ao tools: State of the (aspectj^{TM}) art and open problems. Presentation at the OOPSLA 2002 AOSD Tools Workshop, 2002.

[KL+97] Gregor Kiczales, John Lamping, et al. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Lecture Notes in Computer Science*, volume 3586, pages 195–213, London, UK, 2005. Springer-Verlag.

[KSUH93] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, 1993. CRC Press.

[Lea96] Douglas Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[Lis88] Barbara H. Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto Research Center, Palo Alto, CA, USA, February 1997. Available from: `citeseer.ist.psu.edu/lopes97language.html`.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP 98)*, 1998.

[Low03] Juval Lowy. Decouple components by injecting custom services into your object's interception chain. *MSDN Magazine*, March 2003, 2003.

[LS83] Barbara H. Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, 1977.

[Lum02] Markus Lumpe. On the representation and use of metadata. In *ECOOP '02: Proceedings of the Second Workshop on Composition Languages*, London, UK, 2002. Springer-Verlag.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[May99] Herwig Mayr. *Projekt Engineering. Ingenieurmäßige Softwareprojekt-Entwicklung, 5. Auflage.* University of Applied Sciences Hagenberg, Hagenberg, Austria, 1999. Also published as "Projekt Engineering. Ingenieurmäßige Softwareentwicklung in Projektgruppen" by Hanser Fachbuchverlag, München, 2005.

[Mic05] Brenda M. Michelson. "service-oriented world" cheat sheet: A guide to key concepts, technology, and more—part 1. Technical report, Patricia Seybold Group, Boston, USA, 2005. Available from: `http://www.nextslm.org/michelson1.shtml`.

[Mic06a] Microsoft Corporation. *DynamicMethod Class*, 2006. Available from: `http://msdn2.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx`.

[Mic06b] Microsoft Corporation. *Reflection Emit DynamicMethod Scenarios*, 2006. Available from: `http://msdn2.microsoft.com/en-us/library/sfk2s47t.aspx`.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[Paw05] Renaud Pawlak. Spoon: annotation-driven program transformation — the aop case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible solution for aspect-oriented programming in java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24, London, UK, 2001. Springer-Verlag.

[PSR05] Renaud Pawlak, Lionel Seinturier, and Jean-Philippe Retaillé. *Foundations of AOP for J2EE Development.* Apress, Berkely, CA, USA, 2005.

[PWBK05] David J. Pearce, Matthew Webster, Robert Berry, and Paul H.J. Kelly. Profiling with aspectj. *Submitted for publication to Software: Practice and Experience*, 2005. Available from: `http://www.mcs.vuw.ac.nz/~djp/djprof/index.html#pubs`.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[Ric06] Jeffrey Richter. *CLR via C#, Second Edition*. Microsoft Press, Redmond, WA, USA, 2006.

[Rie04] Johannes Riemer. Corso utilities and collections. Technical report, Vienna University of Technology, 2004.

[Rot05] Andreas Rottmann. Corso serialization for .net. Technical report, Institute of Computer Languages, Vienna University of Technology, 2005. Available from: `http://stud3.tuwien.ac.at/~e9926584/CorsoSerialization.html`.

[Saf06] Dan Saffer. *Designing for Interaction: Creating Smart Applications and Clever Devices*. Peachpit Press, Berkeley, CA, USA, 2006.

[Sch96] Douglas C. Schmidt. A family of design patterns for application-level gateways. *Theory and Practice of Object Systems*, 2(1):15–30, 1996.

[Sel04] Chris Sells. *Windows Forms Programming in C#*. Addison-Wesley Professional, Boston, MA, USA, 2004.

[SG+05] Kevin Sullivan, William G. Griswold, et al. On the criteria to be used in decomposing systems into aspects. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005, Lisbon, Portugal, September 5–9 2005)*, 2005.

[Ske06] Jon Skeet. Implementing the singleton pattern. *C# and .NET articles and links*, 2006. Available from: `http://www.yoda.arachsys.com/csharp/singleton.html`.

[Som01] Ian Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[SP88] A. Silberschatz and J. L. Peterson, editors. *Operating systems concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[SS86] Leon Sterling and Ehud Shapiro. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[sS04] Tomáš Seidmann. *Distributed Shared Memory in Modern Operating Systems*. PhD thesis, Faculty of Informatics and Information Technologies, Bratislava, Slovakia, 2004.

[STW+06] Frans Sanen, Eddy Truyen, Bart De Win, Wouter Joosen, Neil Loughran, Geoff Coulson, Awais Rashid, Andronikos Nedos, Andrew Jackson, and Siobhan Clarke. Study on interaction issues. Technical report, AOSD-Europe, 2006. Available from: `http://www.aosd-europe.net/deliverables/d44.pdf`.

[Sun03] Sun Microsystems. JavaSpaces™ service specification. Technical report, Sun Microsystems, 2003. Available from: `http://java.sun.com/products/jini/2.0/doc/specs/html/js-title.html`.

[Sun04] Sun Microsystems. Build a compute grid with jini™ technology. Technical report, Sun Microsystems, 2004. Available from: `http://www.jini.org/whitepapers/JINI_ComputeGrid_WP_FINAL.pdf`.

[Sut00] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Sut02] Herb Sutter. Standard c++ meets managed c++. *C/C++ Users Journal, C++ .NET Solutions Supplement*, 20(9), 2002.

[tAT03] the AspectJ Team. *The AspectJ™ Programming Guide*, 2003. Available from: `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[tAT05] the AspectJ Team. *The AspectJ 5 Development Kit Developer's Notebook*, 2005. Available from: `http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[Tec04a] Tecco Software Entwicklung AG. *Corso C++ &Co API Documentation*, 2004.

[Tec04b] Tecco Software Entwicklung AG. *Corso Java &Co API Documentation*, 2004.

[Tec04c] Tecco Software Entwicklung AG. *Corso .NET &Co API Documentation*, 2004.

[TG01] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for coordination in web-based systems. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 322–327, Washington, DC, USA, 2001. IEEE Computer Society.

[TOHSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Ver04] Hamilton Verissimo. *Castle's DynamicProxy for .NET*, 2004. Available from: `http://www.codeproject.com/csharp/hamiltondynamicproxy.asp`.

[WeKT00] Heidemarie Wernhart, eva Kühn, and Georg Trausmuth. The replicator coordination design pattern. *Future Generation Computer Systems*, 16(6):693–703, 2000.

[Wel04] George C. Wells. New and improved: Linda in java. In *ACM International Conference Proceeding Series, Vol. 91, PPPJ '04: Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 67–74. Trinity College Dublin, 2004.

[Win05] Wintellect. *Wintellect PowerCollections*, 2005. Available from: `http://www.wintellect.com`.

[WMLF98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998. Available from: `http://www.research.ibm.com/journal/sj/373/wyckoff.html`.

[WT00] Jim Waldo and The Jini Team. *The Jini^{TM} Specifications, Edited by Ken Arnold (2nd Edition)*. Addison-Wesley Professional, Boston, USA, 2000.