FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Technical Criteria for the Productivity of Rapid Web Development Frameworks in Enterprise Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian Thomas
Matrikelnummer 0526537

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Thomas Grechenig
Mitwirkung: Mario Bernhart

Wien, 12.9.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Research Group Industrial Software (INSO)
Institute for Computer Aided Automation (E183)
Faculty of Informatics
Vienna University of Technology

Master of Science Thesis

Technical Criteria for the Productivity of Rapid Web Development Frameworks in Enterprise Java

Author:
Christian Thomas
Braunspergengasse 28/10, A-1100 Wien

Supervisor:
Thomas Grechenig
Mario Bernhart

Vienna, September 12, 2011

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 12.9.2011

CHRISTIAN THOMAS

Abstract

In recent years, the World Wide Web has emerged as a central part of our lives. Accordingly, Web Engineering nowadays is among the main activities in software development. However, due to their special characteristics, developing Web Applications is a complex task. Web Application Frameworks aim at easing this task. On the one hand, the proliferation of these software-tools has made the selection of a framework a challenging task. On the other hand, the competition also led to an ongoing equalization in the market. Today, the existing Web Frameworks hardly differ regarding their functional expressiveness, making it a insignificant criterion when choosing a framework. However, when Ruby on Rails was released in the end of 2005, it unmasked that Web development till then lacked an important factor: Productivity. Since that time, also Web Frameworks in the Java environment are increasingly facing rapid development. Five years after the first version of Rails, this thesis describes the current state-of-the-art in rapid Web development with Java. It analyzes the technical factors that altogether define a Web Application Framework's productivity. The resulting catalog encompasses more than 120 criteria. This way, it can answer the question of which particular Web Framework to choose in an upcoming project. The catalog can also be used to base new comparisons on it and it may even foster further developments in the market.

Zusammenfassung

In den vergangenen Jahren hat sich das World Wide Web zu einem zentralen Bestandteil unseres Lebens entwickelt. Mithin konnte sich auch die Webentwicklung als Disziplin innerhalb der Software-Entwicklung fest etablieren. Aufgrund ihrer speziellen Charakteristika stellt die Entwicklung von Web-Applikationen ein komplexes Unterfangen dar. Web Frameworks sollen diese Aufgabe erleichtern. Die Vielzahl vorhandener Tools macht hier zwar einerseits die Auswahl schwierig, hat inzwischen aber auch zu einer Angleichung der Funktionalität geführt: Vorhandene Web Frameworks unterscheiden sich hinsichtlich ihrer Ausdruckstärke kaum noch, so dass diese nicht länger als Kriterium bei der Wahl des passenden Werkzeugs dienen kann. Die Veröffentlichung von *Ruby on Rails* rückte jedoch schlagartig einen Faktor in den Vordergrund, der von vielen Lösungen bis dahin nicht adressiert wurde: die Produktivität des Entwicklers mit dem jeweiligen Tool. Seitdem haben auch viele Frameworks in der Java-Umgebung agile Entwicklung auf ihre Agenda gesetzt. Fünf Jahre nach dem Erscheinen von Rails untersucht diese Arbeit den derzeitigen Stand der Technik auf dem Gebiet und ermittelt die technischen Faktoren, die die Produktivität der derzeitigen Lösungen definiert. Der resultierende Katalog umfasst mehr als 120 Kriterien. Er erleichtert Managern die Auswahl der passenden Web Frameworks, kann aber auch als Grundlage neuer Web Framework-Vergleiche dienen und die Entwicklung vorhandener Werkzeuge vorantreiben.

Contents

List of Figures

List of Tables

Code Listings

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
AOP	Aspect-Oriented Programming
API	Application Programming Interface
CDI	Context and Dependency Injection
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
COCOMO	Constructive Cost Model
CRUD	Create, Read, Update and Delete
DAO	Data Access Object
DNS	Domain Name System
DOM	Document Object Model
DSL	Domain Specific Language
DTD	Document Type Definition
EAR	Java Enterprise Application Archive
EJB	Enterprise Java Beans
EL	Expression Language
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ITD	Inter-Type Declaration
IDE	Integrated Development Environment
IP	Internet Protocol
GORM	Grails Object Relational Mapping
GWT	Google Web Toolkit
JAR	Java Archive
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JCP	Java Community Process
JDBC	Java Database Connectivity API
JDK	Java Development Kit

JMS	Java Message Service
JPA	Java Persistence API
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JSTL	Java Standard Tag Library
JVM	Java Virtual Machine
GSP	Groovy Server Pages
LGPL	GNU Lesser General Public License
LoC	Lines of Code
MOP	Meta Object Protocol
MVC	Model View Controller
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
REST	Representational State Transfer
RIA	Rich Internet Application
RMI	Remote Method Invocation
SQL	Structured Query Language
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAF	Web Application Framework
WAR	Java Web Application Archive
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSD	XML Schema

Chapter 1

Introduction

In recent years, the Internet, and especially the WWW, has evolved to become a central part in our lives. People nowadays use it to follow the news, compare airfare deals or connect with others in social networks. The same way, companies have discovered the WWW for their matters. Standards set only a few years ago are no longer sufficient today. The shift from static, document-centric Web sites to highly dynamic Web applications raises the demand on skilled developers and made Web development become a main activity in today's software development landscape.

However, the development of Web applications differs significantly from building traditional software products. This relates to both the project characteristics, with usually much shorter timelines, and the technical challenges developers are faced with here. These challenges, resulting from the distributed nature of Web applications and the stateless architecture of the HTTP protocol, have caused a proliferation of software tools that specifically address Web development. Among these tools are the so-called Web Application Frameworks (or Web Frameworks, for short).

Web Frameworks combine a set of software components and specific programming models to a generic functionality that, all alone, allows for the development of full-fledged Web applications. Therefore, developing a new Web application requires choosing a specific Web Framework first. Decisions become more complex by the amount of alternatives at disposal, though. Only regarding Web Frameworks in the Java segment, over 50 known open source solutions do exist. And even by taking only those into account that are used in practice and known to be mature, still more than a dozen remain. Luckily, the heavy competition in the market has led to a broad equalization regarding a Web Framework's expressiveness: It's hard to think of any problems that cannot be solved with one of the popular frameworks.

However, when Ruby on Rails, a Web Framework based on the dynamic programming language Ruby, was released in 2005, it unmasked that Web development in Java, though resulting in quite powerful applications, lacked an important factor: Productivity.

There were too many setup tasks to fulfill, too many configuration files to manage and too many layers of indirection to take care of. Rails revealed that this was by no means a necessity. Nevertheless, the power of Java was still required by enterprises and a simple switch to Rails was not possible for most. But since that time, productivity is also increasingly addressed by Java Web Frameworks: Today, it is the main differentiator between frameworks. Unfortunately, project managers are left alone with the problem of how to decide for or against one of them.

This diploma thesis analyzes the technical criteria that altogether define a Web Application Framework's productivity. As a result, a comprehensive catalog of criteria will be introduced. The catalog will serve different purposes: (1) Five years after the first version of Rails, it will describe the current state-of-the-art in rapid Web development with Java. This information will be especially useful for managers that currently analyze the market and think about adopting a Java Web Framework in their next software project. (2) The criteria can also be used to answer the question of which particular tool to choose in an upcoming project. Currently, the only solution is to follow a prototyping strategy and test multiple projects in parallel. In such scenarios, the catalog will help to focus on the important criteria and objectify the overall selection process. (3) Prototyping and framework analysis could even be completely omitted if meaningful comparisons between Web frameworks would exist. There already exist many such comparisons, but for the most part, they focus on a subset of functional requirements, which, for the reasons mentioned above, usually won't be crucial in a development project. But based on the catalog, new comparisons could be created and identify the most productive Web frameworks. (4) Finally, the catalog may also be useful for Web framework developers themselves. They can identify the shortcoming of their current solutions and close possible gaps.

At the beginning of the thesis, the available literature in the field was studied. It soon became clear that there was no thorough study on the productivity of Web frameworks: Web frameworks in general can be considered a topic rather untouched by science. In contrast, productivity of software development by now is studied for more than 40 years. Thereby, science focuses on two main questions: How the productivity of a project can be measured and estimated, and which general criteria affect productivity. Serious cost estimation models are thereby based on answering the second question first. As according information was still missing for Web Frameworks, this revealed that the work should focus on the productivity-defining criteria.

In a next step, Web frameworks needed to be chosen for a closer examination. The selection process was driven by different factors. To guarantee the significance of the results, a focus was set on the Java platform: Besides being among the most important platforms for Web development, the amount of available Web frameworks reaches its peak here. This indicates that a study on productivity is highly required in this field. Furthermore, each of the frameworks should be used in real development projects and considered to be productive. After an investigation on the Internet and different devel-

opment communities, three frameworks were chosen, namely JBoss Seam, Grails and Spring Roo in combination with Spring MVC. All of them are flexibly licensed under the LGPL resp. Apache License.

The familiarization with a Web framework is considered to be very time consuming. Accordingly, a significant part of the time was consumed by this task. The gained knowledge was then used to create a first structure of the criteria catalog. This structure gave a first impression on which areas to focus on. By studying the respective parts in depth, the catalog was then expanded in an iterative manner, becoming more and more fine-grained over time. In the end, the final catalog contained more than 120 criteria and can be found in Appendix A.

Web Frameworks are built on a variety of technologies. Therefore, to understand the criteria catalog, a basic knowledge of these technologies is a prerequisite. Chapter 2 provides an appropriate introduction, speaks about the special characteristics of Web applications and defines Web Application Frameworks and the philosophy behind them. Productivity, and especially the influence of software tools, is then defined by chapter 3. Related work is discussed and the research questions are derived. Next, the Web Frameworks examined in the course of this work are shortly presented in chapter 4. At this point, the reader should have a basic understanding of the concept behind Web Frameworks and a first impression on how they relate to development productivity. The criteria analysis as the main part of this work is provided in chapter 5. A detailed overview of all criteria found is given by Appendix A. Finally, chapter 6 summarizes the results of the work, gives an outlook on the trends to be expected for the market, and discusses the topics still open for research.

Chapter 2

Fundamentals

A Web application is commonly defined as

> *"[...] a software system based on technologies and standards of the World Wide Web Consortium (W3C) that provides web specific resources such as content and services through a user interface, the Web browser."* [50]

Therefore, to understand the characteristics of Web browser applications and the specifics of their development, section 2.1 first explains the very basic standards that form the Web. Section 2.2 then defines Web applications and the relating concepts in more detail. The Model-View-Controller pattern, a central architectural design in many Web applications and also important for the structure of the criteria catalog defined later in this work, is shortly described in section 2.3. As a preparation for the following chapters, section 2.4 then discusses the most important concepts and technologies relevant to Web Application Frameworks and their usage. Finally, section 2.5 introduces Web Application Frameworks and the philosophy behind them.

2.1 Fundamental Technologies of the WWW

The World Wide Web (WWW) is one of the most important services of the Internet. Using the hypertext system, clients can access documents stored on Web servers anywhere in the world, making distribution transparent [72]. The term 'document' needs to be taken in its broadest sense though [72]: Having been purely static in the beginning of the Web in the early 1990s, a document nowadays may provide dynamically generated content, involving all kinds of active elements like audio, video or client-side scripts for a dynamic behavior.

The location of a document and the way to access it are specified by an Uniform Resource Locator (URL). The Domain Name System (DNS) is responsible to map such human readable URLs to IP addresses [72], that is, to address information the Internet's

underlying Internet Protocol uses to identify and address hosts. To retrieve a document, the user either has to (1) know its URL, (2) follow a hyperlink defined in another document or (3) consult a search engine.

To retrieve and read documents, clients make use of a Web browser. A Web browser is a software application located on the client's machine, typically providing a GUI that eases navigation [72]. The communication between a browser and a Web server and the ways the browser presents information to the user are widely standardized. The most important of these standards are shortly described in the following sections.

2.1.1 Communication-specific Technologies

The *Hypertext Transfer Protocol (HTTP)* protocol has "become the most popular transport protocol for Web contents" [50]. HTTP usually builds on top of TCP on the transport layer. Conversations in HTTP are request/response based, where the Web browser sends a request and a Web server responds. Both request and response consist of a message header and message body. [6]

Each HTTP request contains the URL of the requested resource and the *HTTP method*: The method defines the kind of action the clients invokes on the respective resource. Though there are nine HTTP methods defined, Web browsers mainly use GET and POST [6]. The GET-Method is used to request the resource specified in the URL. Actions implemented by GET ought to be idempotent, to invoke them "over and over again, without unwanted side effects" [6]. The server's state should not be affected by GET (irrespective of logging etc.) [6]. When sending data using GET, it is appended to the requested URL and therefore becomes visible in the Web browser's address bar. This is a common approach to submit data required to handle the request, e.g. to define the keywords on a search engine. One advantage of this solution is that the response becomes bookmark-able. POST's only intention is to send data to the server. In contrast to GET, the data here becomes part of the request's message body and no limitation in size occurs. POST does not need to be idempotent. [6]

A very importing thing to stress is that HTTP is stateless: Associating a HTTP request with a former request of the same client is not possible without any further ado [6]. The statelessness of HTTP is such a big issue that 2.2.1 will discuss solutions to this problem in a separate section.

2.1.2 Document-specific Technologies

There are different presentation technologies Web browsers do support. Many of these technologies (and all which will be discussed here) were developed by the World Wide Web Consortium (W3C), a standards organization for the WWW.

The *HyperText Markup Language (HTML)* was the first technology in this regard. For being a markup language, the documents contain special annotations that need to be distinguished from normal text. In HTML, these annotations are called *tags* and are

enclosed within angle brackets. Tags define how the client's Web browser has to render the document. [17]

Nowadays, HTML documents are often accompanied by Cascading Style Sheets (CSS). By defining multiple style rules in a declarative manner, CSS separates presentation from content. Since the publication of CSS, presentational HTML markup is concerned deprecated. [17]

HTML is "weak in its ability to describe the structure and the meaning of a document's content" [17], though the upcoming version HTML 5 is proposed to put things right here. Up to now, the issue is addressed by another standard, the *Extensible Markup Language (XML)*. XML is a meta-language for markup languages, i.e., XML can be used to define own tags in a new, domain specific language. In contrast to HTML, the structural rules are stricter: Parsers can easily check whether a document is *well-formed*, that is, whether it obeys all structural XML rules and whether it is also *valid*, meaning it adheres to the grammatical, domain-specific rules defined for this document. Such rules are written in a special schema language, usually XML Schema (XSD) or Document Type Definition (DTD) [17]. In software development, XML is a popular way to define configuration metadata (see section 2.4.1). However, regarding presentation, it is most important with respect to the Extensible Hypertext Markup Language (XHTML). XHTML redefines HTML in a way that it becomes valid XML (with all the advantages described above), all by preserving the compatibility with Web browser supporting only plain HTML [17].

2.2 Web Applications

The standards discussed in the last section form the basis of the WWW. However, there is more to Web applications than providing static hypertext documents. Section 2.2.1 will first discuss different enhancements applied to both sides of the client/server architecture, evolving the Web to "a platform able to support the execution of complex applications" [17]. These enhancements have led to broad range of Web applications, described in 2.2.2, before discussing the characteristics that make Web applications and their development distinct from common software (section 2.2.3).

2.2.1 Towards Dynamic Web Pages

The WWW was originally intended to provide simple static hypertext documents to the user. Due to its huge acceptance, new requirements arose quickly, steadily evolving the Web to the application platform we know today [17]. Some of the major enhancements are discussed in this section.

Application execution engines

An early approach towards dynamic pages was the Common Gateway Interface (CGI), which had some significant drawbacks regarding performance and state management.

These limitations were solved by application execution engines [17] like Java Servlets, which will be discussed in-depth in section 2.4.2. Like CGI, they allow to call arbitrary program code, for example, to query or update a database. However, by computing HTTP responses using application execution engines, process creations and terminations are omitted and resources can be shared easily in a concurrent manner. These tasks are usually managed and optimized in the background; the developer no longer has to think about it [17]. And because the use of processes is reduced, data can easily be kept in a single place, making also the implementation of session management easier (see below).

All the mentioned points were common problems of the old CGI.

Server-side scripting

Servlets completely consist of programming instructions. Regarding the document-specific technologies discussed earlier in this chapter, this is an unnatural way of generating user response. *Server-side scripting* allows the insertion of program code into page templates which, apart from that, look like usual hypertext documents, as for example HTML. To deploy such scripts, the Web server must be equipped with the respective scripting engine [17]: When a request arrives, the Web server passes the page template to the scripting engine, which then processes the programming instructions. The result sent to the user is a plain hypertext document. One representative for this approach is the JavaServer Pages (JSP) technology, discussed later in section 2.4.3.

Session management

Since HTTP is a stateless protocol (see section 2.1.1), this raises the question of how retaining state across multiple user requests is possible nonetheless. There are various workarounds to this problem; the two most important are discussed here. The first solution works using *cookies*.

> *"A cookie is an object created by a server-side program and stored at the client (typically, in the disk cache of the browser), which can be used by the server-side program to store and retrieve state information associated with the client."* [17]

In its response, the server can send a Set-Cookie HTTP header to create a cookie filled with the information in the value part of the header. The client will always transmit this data in following requests. This way, the server can deposit client and session identifiers in the cookie and can easily access this information in future requests. [6]

Since cookies can define arbitrary URLs they are valid for and get send to – absolutely independent of the URL actually visited – there has been an abuse of this technology, concerning tracking of user behavior and advertisements (see [34] for details). As a result, many clients do not accept cookie creation or may delete them unpredictably [6]. Therefore, relying on cookies is not enough.

URL-Rewriting as a second solution does not require cookies. Using this approach, the client's state information simply gets appended to all URLs the user requests on the

respective website. However, this is no action supported by the browser: The application must take care of rewriting each link created in any of the dynamic pages the user requests [17]; many technologies, like for example Servlets, take this work from the developer automatically though. One real drawback of this solution is the misuse of the URL for state management, bad if the developer is a friend of a RESTful[1] URL design.

Java Servlets (see section 2.4.2) combine both approaches in the following way: The very first response sent to a client uses both methods, cookies and URL-Rewriting. If the cookie data gets send back in the following request, URL rewriting is stopped for the rest of the session. This way, the Servlet technology allows for session management and also provides clean URLs if possible. [6]

Client-side scripting

Over the years, enhancements have not only been applied to the Web server. With the Web evolving, new presentation and interactivity requirements arose [17].

JavaScript is a client-side scripting language executed by the Web browser [17]. In the context of this work, it is important to stress that, irrespective of its name, JavaScript has nothing in common with Java [6]. The JavaScript code is either part of the HTML markup or comes in external files. It may be invoked when the page is loaded or an event from one of the HTML elements is raised because of a user interaction. Today, JavaScript is the only scripting language "adequately supported by most of the Web browsers on the market" [17]. Though being the de-facto standard, there are still browser-specific peculiarities a developer has to take into account. Tools like the Google Web Toolkit (GWT)[2] provide a higher-level approach towards dynamic page presentation: The framework generates JavaScript code and HTML markup from common Java code, handling all the stated peculiarities automatically [43].

Leveraging the Document Object Model (DOM), JavaScript is able to modify a HTML document dynamically.

> *"DOM supplies a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them."* [17]

Using JavaScript, DOM and the `XMLHttpRequest` API it even becomes possible to transparently perform HTTP requests in the background, dynamically adding the received content to the page. This approach is also known as Asynchronous JavaScript and XML (AJAX). AJAX allows for new interaction possibilities and highly responsible pages. [17]

[1] REST = Representational State Transfer. A movement having its orgin in the Web service field, that, among other things, advocates a more natural usage of HTTP and URLs that are only used to identify resources.

[2] Project Site: `http://code.google.com/webtoolkit`

2.2.2 Categories of Web Applications

At this point of the chapter, it should already be apparent that different kinds of Web applications exist: Some might return purely static resources, while others provide access to sophisticated business logic in a highly dynamic manner. The different categories are depicted in Figure 2.1. Obviously, there is a correlation between development history and degree of complexity. However, a newer category is not necessarily a full replacement of an older one. The same way, a Web application may also fall into several categories at once. [50]

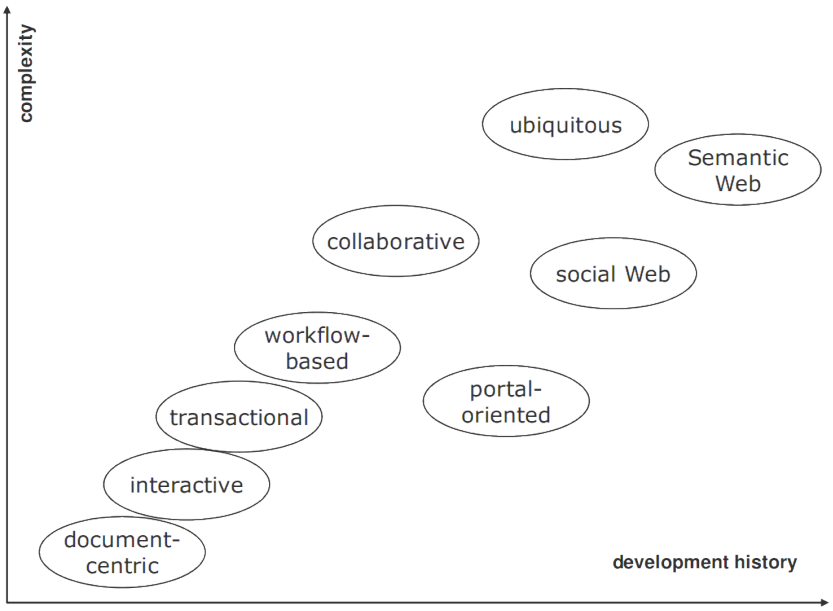

Figure 2.1: Categories of Web applications [50]

The following is just a short overview on the different categories. For a detailed description, see [50].

DOCUMENT-CENTRIC The Web site only provides access to plain hypertext documents.

INTERACTIVE Interactivity by means of HTML forms is provided; pages may be generated dynamically according to user input.

TRANSACTIONAL Performing updates on the underlying database becomes possible.

WORKFLOW-BASED These applications allow the handling of workflows, typically based on Web services.

COLLABORATIVE In contrast to workflow-based Web applications, this category is able to deal with unstructured operations. Communication between users is important here.

SOCIAL WEB Here, users provide their identity to a community of other users with similar interests.

PORTAL-ORIENTED These applications offer a central point of access to separate and possibly heterogeneous sources of information and services.

UBIQUITOUS As the name says, this category of application facilitates ubiquitous access, independent of location or device.

SEMANTIC WEB The provided information aims at being understandable and usable by machines, e.g. to facilitate knowledge management.

2.2.3 Characteristics of Web Applications

Web applications differ from usual software due to characteristics which aren't of particular importance or do not even exist in traditional software products [50]. The existence of these characteristics is what makes Web development so special and is the reason why existing processes, procedures and tools need to be adapted for building Web applications [58]. The following section will briefly examine these characteristics (for a more detailed discussion, see [50]). Obviously, the degree to which a certain characteristic is present also depends on the category of the particular Web application.

Product-related Characteristics

The following factors make a Web application special as a product: (1) The content, (2) the hypertextual structure of its documents and (3) the User Interface.

Content is the main argument for using a particular Web application. Content must be generated, made available and updated frequently. The best way to present the information also depends on the respective user group and its quality demands that need to be satisfied. Information needs to be "up to date, exact, consistent and reliable" [50]. Lately, solutions also tend to be location-aware and personalized.

The *hypertextual structure* of documents implies the integration of special techniques to avoid "disorientation and cognitive overload" [50], since each user has an individual style of reading and may move freely within the provided content.

Concerning the *User Interface*, the "high competitive pressure on the Web" [50] makes the "look and feel" a factor that determines upon success or failure of the application. In particular, the User Interface should be self-explainable, for often, one cannot expect the user to read documentations or visit training courses.

Usage-related Characteristics

The usage of a Web application may vary concerning (1) the social context, that is, the specific user, (2) the technical context, defined by the network connection and the accessing devices and (3) the natural context, meaning location and time. Predicting these factors is hardly possible which implies the "necessity to continuously adapt to specific usage situations" [50].

For a *user*, it is easy to find competing applications in the Web. That is, application usage is always spontaneous which has implications on scalability and reliability

requirements [58]. Furthermore, the heterogeneity between users is of a high degree concerning their abilities, knowledge and preferences.

Concerning the *technical context*, there are multiple factors developers cannot influence and therefore have to make assumptions about. E.g., bandwidth and reliability of the connection have a big impact on the provided quality of service; devices that access the application may differ widely in their specifications. Additionally, users can configure their Web browsers autonomously which also influences functionality.

Last, Web applications may be accessed from any geographical position at any time (*natural context*) which implies new requirements concerning internationalization and security, but also provides new opportunities regarding location-aware or time-aware services.

Development-related Characteristics

The development of Web applications is special in terms of (1) the development team, (2) the technical infrastructure, (3) the development process and (4) integration concerns.

In comparison, *development teams* in Web application engineering have a young average age [50], thus making the team less experienced. Furthermore, Web development should always take a multidisciplinary approach, based on expertise from several areas: In addition to IT experts, there should also be experts regarding hypertext, design and the specific domain.

Inhomogeneity and immaturity characterize the *technical infrastructure*. Web servers can easily be configured, but this does not hold true for the client's Web browser. Furthermore, components used in Web development often have bugs or are updated frequently, requiring a change of the development environment as well.

Concerning the *development process*, flexibility is very important. Web development projects cannot adhere to a predefined project plan [50]. Furthermore, short development times and the fact that Web applications often consist of autonomous components allow for parallel development by various subgroups.

Last, regarding integration, there often is a need for *external integration*: Web applications may be based on content and services provided by external applications the development team is not in control of and has only few knowledge about.

Evolution

Evolution is what guides all the three dimensions discussed before [50]. Web applications are always subject to continuous change. Because of the high competitive pressure and the rapid change on the Web with users always wanting the "newest Web hype" [50], development cycles have become shorter. However, as surveys show (see [58]), time-to-market, unlike traditional software engineering, is not the most important quality process driver: To be competitive, it is often advantageous to be "later and better" than "sooner but worse".

2.3 The Model-View-Controller Pattern

The Model-View-Controller Pattern (MVC) is an architectural pattern in the context of interactive applications providing a user interface. MVC

> *"[...] divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input."* [16]

According to [36], what is most important about MVC is the separation of presentation from the model. User interfaces tend to change quite often [16]. Furthermore different users may exist, having conflicting requirements on the user interface but not on the logic behind, that is, the model. This separation should always be followed, except the simplest systems where the model has no real behavior. Applying MVC, the model is independent of the User Interface. If the developer also implements the second separation between view and controller, then the model is also independent of specific input behavior. However, this is concerned less important, since most systems have only one controller per view anyway [36]. Not implementing this separation results in the *Document-View* pattern [16].

MVC is visualized in Figure 2.2.

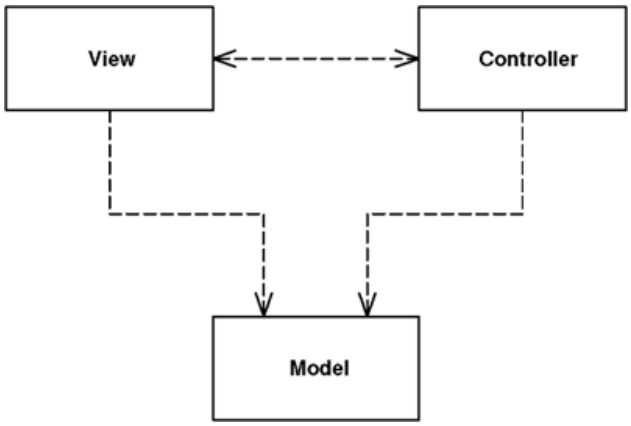

Figure 2.2: The Model-View-Controller Pattern [36]

For Web development, there exists the so called *Model 2* architecture, which can be regarded as a server-side implementation of Model View Controller (MVC) [68]. In the Model 2 architecture, a Servlet is used to process the request. Depending on the request, the Servlet may instantiate the model. The Model 2 pattern does not clearly define the model; it simply can be any data-structure fulfilling the developer's needs. The Servlet then forwards the model to a JSP, which is now responsible to generate the view based on the model's data. The JSP does no processing at all. [68]

The original MVC pattern postulates the use a change-propagation mechanism between the view and the model, based on the Observer-Pattern (see [16] for details). This

is the reason why there are voices (e.g., [29]) that Model 2 is a new way of parameter passing, but no implementation of MVC.

2.4 Specifications and Technologies in Enterprise Java

Information technology has become an important issue for today's enterprises. They need to deliver services to a broad range of users and therefore have high demands on the availability, reliability, scalability, security and other qualities of these services. By providing a platform for server programming, Java Enterprise Edition (Java EE) aims at reducing the development effort for such enterprise services [19].

However, there are also specifications or technologies outside the official EE platform that are very important in enterprise development with Java: Some are part of the Java Standard Edition (Java SE) aiming for general programming use, others aren't even part of an official specification but are nevertheless used extensively, like the Spring Framework (see section 2.4.8). The (unofficial) term *Enterprise Java*, as used in the title of this master thesis, comprises all these technologies.

In the following sections, the most important specifications and technologies in Enterprise Java will be introduced. This introduction will not exceed the amount of information required to understand the following chapters though.

2.4.1 Important Concepts

Technologies concerning Enterprise Java make use of a few typical programming concepts and paradigms not necessarily known to developers in other application fields. These concepts are also important for understanding the following chapters of this work.

Metadata

Java Enterprise technologies often require additional information associated with certain framework or programming elements. Since these elements still need to be interpreted by a Java Virtual Machine (JVM), they need to be defined in a way that does not interfere with the usual programming syntax. [56]

Basically, there are two common solutions for this problem: Either the developer defines the metadata in an external file, usually written in XML, or the developer uses *annotations*. Annotations were introduced in Java 5 as a way to define class metadata directly in the source code [56]. Examples for such annotations in Java SE are `@Override` or `@Deprecated`. They also play an important part in the Convention over Configuration principle of modern Web Applications Frameworks (see section 2.5.2).

Both XML and annotations have their pros and cons. Annotations usually provide a better usability. Because they are defined in the source code, they are more documental than XML and a configuration in question is easier to find. Furthermore, annotations are type-safe. When using XML, adaption requires changes on multiple files. However,

configuration changes require no recompilation of the source code. And to handle general configuration issues, valid not only for single classes, using external definitions is the only option. [54]

Many technologies allow both types of metadata definitions, often letting XML configurations overwrite existing annotations. In such cases, development teams should define conventions when to use which mechanism.

Inversion of Control and Dependency Injection

Inversion of control is often referred to as the Hollywood Principle – *"Don't call us, we will call you"*. It is a key part of what makes a framework differ from a library [37]. Inversion of control is the

> *"[...] characteristic of a framework [...] that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework [...] plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application."* [48]

In the course of this work, it is important to stress that inversion of control is also inherent to every Web Application Framework.

Dependency injection is a special form of the Inversion of Control principle used to let frameworks inject references to required resources into class instances. This decouples the different components, resulting in a better application design which is easier to configure and easier to test [74]. There are different forms of dependency injection: References may be injected into fields directly, `set` methods can be used or the references get bound in the object's constructor. Setter Dependency Injection gives more control over the injection process, while Constructor Dependency Injection makes sure that the respective dependencies are satisfied after the object's initialization. The different injection points are either annotated or get defined using external configuration files.

2.4.2 Java Servlet Technology

The Java Servlet Technology provides a request-response programming model. The request could be of any type, but commonly Servlets communicate using HTTP. [6]

At runtime, there usually exists only one instance per Servlet definition. Each request is handled in an own thread. This approach offers a significant performance gain; however, the developer has to keep concurrency issues in mind and synchronize appropriately where required. [6]

Servlets are defined in the `javax.servlet` and `javax.servlet.http` packages. Usually, programmers define own Servlets by extending them from `HttpServlet` which already provides an implementation of this interface. The developer only needs to override one or more service methods that correspond to the HTTP methods defined above in section 2.1.1. There, information from the request can be extracted and external resources can be accessed. The response is either returned directly from within the method or delegated to another Web component, for example, to a JSP. The following code snippet in Listing 2.1 shows a simple Servlet responding to HTTP-GET requests. Package imports and exception handling has been omitted for simplicity.

Listing 2.1: A simple Servlet handling HTTP-requests (`SimpleServlet.java`)

```
1 @WebServlet("/simple")
2 public class SimpleServlet extends HttpServlet {

4   protected void doGet(HttpServletRequest request,
        HttpServletResponse response){
5     response.setContentType("text/html");
6     PrintWriter out = response.getWriter();
7     out.println("<html><body>Hello, World!</body></html>");
8   }
9 }
```

A `HttpServlet` is by no means restricted to HTML in its response. Furthermore, the depicted approach to print HTML pages into the response stream is only recommended for the simplest cases [6]. The next section will show a better solution to this problem.

The `HttpServletRequest` can be used to read out the request, such as request attributes, headers, or the request body. Accordingly, `HttpServletResponse` provides ways to generate a response, like adding headers or a message body. The current user session, basically a map to store and retrieve key/value pairs valid for the whole session, can also be retrieved.

The Java Servlet Technology also defines an event system applications can subscribe for and allows defining filter-chains to adapt requests before they are processed by the Servlet. All this makes Java Servlets highly flexible and is the reason why actually most Web Application Frameworks are based on it. Accordingly, improvements in the latest 3.0 specification for Java EE 6 address easier configurations and a modularization support [47].

2.4.3 JavaServer Pages

JavaServer Pages (JSP) are an example for a server-side scripting language already introduced in section 2.2.1, that is, they focus on the View and consist of static data like HTML and JSP elements [6] which define the dynamic content of the page. Listing 2.2

shows a simple JSP file containing a directive, a declaration, a scriptlet and two expressions.

Listing 2.2: A simple JSP using 4 kinds of JSP elements (`SimpleJsp.jsp`)

```
1 <%@ page import="java.util.*, java.text.*" %>
2 <%@ taglib prefix="my" uri="customTags" %>
3 <html>
4 <body>
5 <%! int count = 0; %>
6 <% SimpleDateFormat sdf = new SimpleDateFormat("HH:mm");
7    String time = sdf.format(new Date());%>
8 You are the <%= ++count %>. user of this page.<p/>
9 Current time is <%= time %>.
10 </body>
11 </html>
```

In the end, the Web container translates and compiles each JSP file into a common Servlet: The static data is directly written to the `HttpServletResponse` object's response stream; JSP elements get transformed into special code constructs [6]. Inside a JSP file, the developer can make use of so-called *implicit objects* that automatically get injected. For example, there are objects representing the `HttpServletRequest`, the `HttpServletResponse` or the current user session.

However, using scriptlets, declarations or expressions in JSP files is concerned a bad practice. Those kind of files are hard to maintain and can hardly be understand by Web designers that need to edit them. Instead, developers are encouraged to use Expression Language (EL) which offers a way to call normal Java code from inside JSP and still appears more natural to non-programmers. By using standard action tags as defined by the Java Standard Tag Library (JSTL) or by defining custom tags, coding in JSP becomes obsolete [6]. The according functionality is implemented using common Java code.

It is also possible to write JSP files in well-formed XML. Those files are called *JSP Documents*.

2.4.4 Bean Validation

Validation of data is a recurring task, important for any application dealing with user input [9]. The Bean Validation specification defines a metadata model to address this issue, relying on annotations or XML. It can be used on any tier and is not restricted for a use in the Java EE environment. [9]

Bean Validation works by applying constraints that consist of two parts: A constraint annotation and constraint implementation. Constraints can be applied to fields, methods, whole classes or even other constraints. The same type can be subject to multiple constraints. The specification already comes with many built-in constraints for various use cases. [9]

An example for how to use the Bean Validation API is given in the following section.

2.4.5 JavaServer Faces

According to the specification, JavaServer Faces (JSF) "is a user interface framework for Java Web applications" [8]. By mapping HTTP requests to reusable and stateful components, it provides a separation of behavior and presentation according to the MVC pattern. The latest release of JSF was version 2.1.

Since the Seam Frameworks (see section 4.1) in its common usage is based on JSF [4], the programming model will be explained in more detail here. The following example includes two pages and a simple component: The first page (in listing 2.3) defines a form, allowing the user to type in her/his first name and send it to the server by pressing the button. After that, some server-side logic executes and finally forwards the user to a second page, printing a personalized greeting. If the provided username is too short or too long, the form page is displayed again, showing an appropriate error message.

Listing 2.3: A facelet defining a form (`input.xhtml`)

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2       xmlns:h="http://java.sun.com/jsf/html">
3 <h:body>
4   <h:messages/>
5   <h:form>
6   Enter your name:
7   <h:inputText value="#{greeting.name}"/>
8   <h:commandButton value="Greet me!"
        action="#{greeting.doGreeting}"/>
9   </h:form>
10 </h:body>
11 </html>
```

In contrast to older versions, *Facelets* are now the preferred presentation technology in JSF [8]. The reason is about to be discussed later in this work (see section 4.1.2). Documents need to be valid XML in Facelets [45].

The prefix `h` is a reference to the HTML JSF components used in the page. JSF defines multiple tag libraries and also allows the developer to define custom tags [8]. Both the input field and the button set a reference to a bean called `greeting`. Listing 2.4 shows the definition of this bean.

Listing 2.4: ManagedBean with CDI and Bean Validation annotations (`Greeting.java`)

```
1 @Model
2 public class Greeting {

4   @Size(min = 2, max = 25)
5   private String name;

7   public String getName() {
8      return name;
```

```
9    }

11   public void setName(String name) {
12      this.name = name;
13   }

15   public String doGreeting() {
16      return "greeting";
17   }
18 }
```

Except the two annotations, `Greeting.java` is a common Java class definition. `@Model` is a CDI annotation: Context and Dependency Injection (CDI) is a new part of Java EE 6 and offers dependency injection for so called *Managed Beans*, that is, objects with an enhanced lifecycle that depend on the current context of the injecting instance. "Clients [...] executing in the same context will see the same instance of the bean" [51]. Besides the loose-coupling offered by dependency injection, developers profit from this approach because they no longer have to think about lifecycle and thread management regarding stateful objects [51]. CDI also brings the concept of *interceptors* to Managed Beans, so this is no longer a reason to use Enterprise Java Beans (EJB) (defined in section 2.4.7). CDI will also play an important role in the upcoming version of the JBoss Seam framework.

Here, `@Model` creates a request-scoped bean and makes it available under a default EL name, which is the unqualified class name, uncapitalized [51]. Furthermore, the class defines a property called `name`, which is annotated again: `@Size` is a built-in annotation from the Bean Validation specification that was introduced in the last section. Finally, there is an action method defined called `doGreeting()`. In this case, the action method forwards the user to the `greeting` view, which, by making use of the provided name, simply prints a personalized welcome message:

Listing 2.5: Facelet printing output (`greeting.xhtml`)

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2       xmlns:h="http://java.sun.com/jsf/html">
3 <h:body>
4   Hello, #{greeting.name}!
5 </h:body>
6 </html>
```

To implement the depicted behavior, JSF processes a request in multiple phases. Depending on the kind of request and the kind of the requested resource, these phases differ widely. The most common lifecycle, however, is where a Faces Request generates a Faces Response [8]. Here, processing works as follows [4]:

(1) Restore view: The component hierarchy is restored using the stored state information from the last request.

(2) Apply request values: From the information in the request, each component can now update its state (the so called *local state*).

(3) Process validations: Next, validations are performed. A short-circuit to phase 6 may occur due to failing validations.

(4) Update model values: The local state can now be used to update the application model.

(5) Invoke Application: The respective action method is called and control is passed over to the application. Either the method ifself or the rules defined in *faces-config.xml* determine the next view to display.

(6) Render response: Last, the response is rendered to the client and the state is stored for following requests.

Further details of the JSF lifecycle can be found in [8].

Regarding previous versions, JSF 2 provides a much better support for the Convention over Configuration principle. Many XML configurations can now be replaced by annotations or even do not need to be specified because reasonable defaults apply.

2.4.6 Java Persistence API

Concerning persistency, there has always been a clash between the object-oriented world on the one side and relational databases on the other. The object-relational mapping approach, as used by the Java Persistence API (JPA), aims at bridging this gap the most transparent way [56]. JPA consists of four different parts: (1) The API, (2) a query language similar to SQL but in an object-oriented way, (3) the Criteria API to find stored data using query-defining objects and finally (4) the object/relational metadata [47].

A very important concept in JPA is the *entity*. According to the specification, *"an entity is a lightweight persistent domain object"* [26]. This object is very similar to a common object in Java, there only exist a few additional requirements. Most important, the object must be configured as an entity and needs to include an ID field and a non-private (empty) default constructor. An entity represents a table in a database, whereas an instance of this entity is stored in a single row of this table. Concerning the entity's fields or properties, different types are allowed, including all primitive types and their respective wrappers, `Date`, `BigDecimal` and other entities. The latter case, an entity being part of another entity, indicates a relationship between these objects. Relationships can either be uni- or bidirectional and have either a 1:1, 1:n or m:n multiplicity. Inheritance is also supported. The JPA implementation automatically deals with mapping these relationships to the database and creates mapping tables where required [56].

Listing 2.6 shows an exemplary `User` entity with a m:n relationship to other users.

Listing 2.6: A simple persistent entity (`User.java`)

```
1 @Entity
2 public class User {

4   @Id @GeneratedValue
5   private Long id;

7   private String username;

9   @ManyToMany
10  private List<User> friends;

12  public User(){}

14  public User(String username, List<User> friends) {
15    this.username = username;
16    this.friends = friends;
17  }

19  // getters & setters
20  ...
21 }
```

Another important part of the JPA specification is the `EntityManager`. As the name says, it manages entities and their lifecycle. At any point of the application, each entity is either in a transient, managed, detached or removed state. Different methods of the `EntityManager` make them change their state. Concerning an entity's state, it is also important whether the `EntityManager` is used in a transaction-based or extended `PersistenceContext`. Depending on an application- or container-managed approach, the default varies here. The `EntityManager` is also used to create queries, set locks on the database or run transactions. [56]

2.4.7 Enterprise Java Beans

The EJB specification defines a component-model for the server-side business logic of an application [47]. An enterprise bean requires an EJB container to run in. This container aims at transparently providing services such as concurrency handling, transactions or security. The idea behind this approach is that the developer can instead focus on the implementation of the core business logic [54].

The central element of EJB is the enterprise bean. This is a component encapsulating business logic. There exist two types of enterprise beans: Session beans and message driven beans. The type also defines the way the container optimizes their management.

"A session bean represents a transient conversation with a client" [47]. How this conversation looks like depends on the actual type of session bean. *Stateless session beans* get assigned to a client for the length of one method call. In contrast, *stateful session beans* have a client-specific state and exist until the client removes the bean or

it times out. *Singleton beans* are new to EJB 3.1 and are only instantiated once per application.

Message Driven Beans allow for an asynchronous communication with clients. This is typically accomplished by using the Java Message Service (JMS) technology: Asynchronous communication decouples senders and receivers, making them independent from a technical perspective. This is a possible approach towards the enterprise application integration. [47]

Clients may access enterprise beans in multiple ways. This depends on whether the respective enterprise bean is defined *local* and/or *remote*. Local communication is possible where clients run in the same application as the enterprise bean. Remote beans can be accessed from clients on different machines or different JVMs and therefore also offer ways to decompose the server-side application, e.g., to introduce load balancing. Communication is based on RMI here. Stateful session beans may also define a Web Service interface. Message driven beans always communicate remotely. [54]

The EJB specification also offers a rich model to implement callbacks, interceptors or define timer-based operations carried out by enterprise beans.

2.4.8 The Spring Framework

In contrast to the technologies discussed beforehand, the Spring Framework[3] does not originate from the official JCP.

Spring became popular in the times of J2EE, that is, before Java EE 5 was released in 2006. Back then, enterprise development with Java and especially the EJB 2.x technology was considered complicated and unproductive.

Spring simplifies the development process by providing a unified abstraction over many Java SE and Java EE APIs and other open source frameworks [74]. It aims at being a non-intrusive framework, providing many optional modules for different development concerns. In contrast to an EJB application, it is possible to deploy a Spring Web application on a common Web server without having to abandon services like container-managed transactions. Long before CDI, Spring delivered a framework for Dependency Injection and also provided help for Aspect-Oriented Programming (AOP) – something that is still not part of the official Java specification. As a consequence, Spring nowadays is considered a de facto Standard for Java enterprise application development [74].

However, because Java EE 5 and Java EE 6 tackled many of the issues stated above, there are voices (like, for example [55]) saying that Spring is no longer needed today. This view obviously disregards that, in the meantime, there is more to Spring than Dependency Injection or being a replacement for EJB: The Spring stack comprises many projects there is no standardized alternative to. Spring Roo, discussed in section 4.3, is one of those projects.

[3] Project Site: `http://www.springsource.com`

2.5 Web Application Frameworks

Because of the special characteristics of Web applications (see 2.2.3), their development is a challenging task. A Web Application Framework (WAF) can fundamentally ease parts of this task. In this section, WAFs will be defined (section 2.5.1), their underlying philosophy will be discussed (section 2.5.2) and a taxonomic scheme will be depicted that allows the classification of different types of WAFs (section 2.5.3). Finally, the main factors that drive the selection of a WAF are named in section 2.5.4.

2.5.1 Definition

The idea behind Web Application Frameworks is no phenomenon unique to Web development. Application Frameworks have been around for many years now. Web Application Frameworks are just a special manifestation for the Web, which is why they share most of their characteristics and motivations. Therefore, before talking about Web Application Frameworks and their strength, the general term of an Application Framework will be discussed first.

Application Frameworks in General

> *"A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications."* [17]

Instead of libraries, which only provide just a single building block for the application under development, Application Frameworks "leverage the sameness between applications" [40] by offering a "skeletal structure for an application solution" [69]. A library provides just a set of functions, which in Java are organized into classes. The developer may or may not call these functions, and after each call, control returns to the client. The behavior of Application Frameworks is fundamentally different: For a framework is a semi-complete application, waiting for the developer to turn it into something useful by adding application-specific behavior, it is the framework itself which calls the developers code. And after performing some work, control is automatically handed back to the framework [37]. This is exactly the behavior defined as "Inversion of Control" in section 2.4.1 on page 14.

Implementing enterprise applications is a complex and error-prone task. The resulting software needs to be correct, efficient and inexpensive. Solving this task "from scratch" is therefore usually no option. This is where Application Frameworks come into play. [32]

Application Frameworks are typically created by experienced developers and based on their profound domain knowledge [32]. This way, using an Application Framework may happen for the same reason programmers are advised to implement design patterns in their code: For recurring problems (which, in this case, is application development in the specific domain) they provide a solution known to work. By using an Application

Framework as a foundation for own products, the re-creation and revalidation of common solutions is avoided. Besides better software quality and performance, their usage also leads to substantial improvements concerning developer productivity. [32]

Application Frameworks all follow the same principles: Using multiple layers of indirection, they all define frozen spots and hot spots. Frozen spots relate to the overall architecture of the framework and its basic components. These remain unchanged by the framework user, that is, they are "frozen". Hot spots, on the other hand, define the parts of the framework that may be extended by the application developer and are specific to the individual application. To take this approach, Application Frameworks are based on the Inversion of Control principle (see section 2.4.1 on page 14 for details). [69]

However, using an Application Framework also suffers from a few weaknesses: First of all, much learning effort, like weeks or months, may be required to really become productive with the tool. This effort usually needs to be amortized over many projects; otherwise, it will likely not pay off [32]. Furthermore, the reliance on Inversion of Control typically makes applications harder to debug. Often, it is not even clear whether a certain bug relates to the application or the framework itself [32]. Last but not least, Application Frameworks are common software; they are subject to changes and need to be maintained. This way, using an Application Framework introduces new dependencies to the project [32].

Web Application Frameworks

A Web Application Framework (WAF) as a special type of Application Framework is commonly defined as

> *"[...] a reusable, skeletal, semi-complete modular platform that can be specialized to produce custom Web applications."* [69]

Because a WAF is an Application Framework, it has the same characteristics and provides the same advantages to the developer; it is only specially aligned for the Web. As such, a WAF includes all the building blocks essential for constructing Web applications. It is usually based on the MVC pattern and supports a n-Tier architecture.

Integrating libraries commonly proven for Web development (an approach known as "best of breed", see section 2.5.2), a Web Framework already provides all the tools to implement the functionality typically required by Web applications. [69] These requirements usually cover libraries for User Interfaces and persistency, user and permission management, form handling and input validation or security, to name but a few. [73]

Some authors ([46]) differentiate between *Web Application Frameworks* and *Agile Web Application Frameworks*. According to them, a WAF is a framework that specializes for a single tier (like persistency or User Interfaces) and only Agile Web Application Frameworks provide full stack solutions. Following this definition, any of the tools examined in this work actually are Agile Web Application Frameworks. However, for simplicity reasons, this work will refer to them as *Web Application Frameworks*.

2.5.2 Design Philosophy

A modern WAF is usually driven by a few core design principles which are discussed in the following.

Convention over Configuration

Convention over Configuration is a design paradigm for software applications. Frameworks designed according to this principle reduce the configuration overhead of the underlying application. This becomes possible by choosing defaults instead of requiring the developer to configure each single detail concerning the application behavior. Framework developers can usually anticipate the most common usage of their tool; therefore, they can easily find sensible values and make a deviation the exceptional case. However, if conventions fail, the framework still needs to provide a way to apply a different configuration (that is, it is Convention *over* Configuration but not Convention *instead of* Configuration). [71]

The main advantage of this principle is that application developers become faster. They do not need to configure every detail, which is a big relief since many configuration errors do not become obvious before runtime. It is easier to maintain the application, for less explicit configuration exists. Furthermore, a modification in the code does not require a modification in the configuration. [18]

However, the Convention over Configuration principle also carries the inherent danger of developers not using the framework with the required care. They overlook that they still need to be familiar with the default configuration values, otherwise, they cannot notice when a deviation from the defaults is required. Finding such errors can become complex, because the application often seems to work perfectly on the first sight. [18]

Templating and Scaffolding

Based on templating and scaffolding, certain development tasks can easily be automated [71]. This way, developers can create Web applications ready to deploy in a few minutes, just by providing a few information about the application to develop (e.g., the project name or a database connection). The WAF then carries out all necessary configurations. For example, required libraries are automatically integrated, the application is ready to use internationalization, database configuration is completed and build files are available. Building such a system manually would often take hours or days, and only be possible to experienced users. Novice users do not have to learn the framework the hard way and can already start developing. Furthermore, the created application architecture usually obeys the best practices for the specific WAF.

Integration and Best-of-Breed

Web applications share some common functional requirements, some of which were already mentioned in section 2.5.1. Quite often, these requirements are already solved

by specific libraries specialized for the respective tasks. Modern WAFs therefore try to delegate most work by integrating these libraries and configuring them beforehand. A common term in this regard is "best of breed", meaning the framework builds on those technologies that have proven in practice and are known to work well together [4], instead of providing own mechanisms that compete with existing good solutions [58].

However, for most Web applications there are also special requirements on technologies that WAF developers cannot anticipate. Therefore, WAF developers have to design the framework in such a way that an integration of new technologies is easy and the framework does not interfere with them. [58]

2.5.3 Taxonomy

Though most WAFs are based on the MVC pattern, there are still different types of Web frameworks, varying in the programming model they promote.

REQUEST-BASED FRAMEWORKS Request-based WAFs stick close to the already mentioned CGI or Servlet approach. Programming in such a framework is quite natural and linear: Controllers receive a request which the developer then processes in a first step. On the basis of the request, the developer invokes some application logic (e.g., information gets stored or is received from the database) and in the end, a view is returned. The application flow is therefore programmed explicitly. Likewise, state is handled in a more explicit manner. [69]

COMPONENT-BASED FRAMEWORKS Component-based Frameworks aim at reusability. The request-processing is abstracted away; the application logic is encapsulated in components and possibly becomes reusable for similar use cases. State is handled automatically by the framework. By implementing event-handling mechanisms, component-based frameworks are closer to the programming model known from traditional GUI applications. [69] The abstraction of this approach results in a more complex and less obvious application flow. E.g., the life-cycle of JSF, which is representative of a component-based architecture, decomposes a single request in up to six phases. [4]

RIA-BASED FRAMEWORKS A Rich Internet Application (RIA) is a Web application which utilizes "asynchronous techniques such as Asynchronous JavaScript and XML (AJAX) to enable real time data communication and dynamic rendering, which give the Web interface a more desktop-like application feel" [31]. RIA-based Frameworks generate a client-side application with state and user interaction. This type of WAF is out of scope of this master thesis for they do not follow a server-centric approach.

In the course of this work, three WAFs will be discussed in detail. JBoss Seam (section 4.1) is an example of a component-based framework whereas Grails (section

4.2) is a representative for a request-based WAF. Spring Roo (section 4.3) will be used in conjunction with Spring MVC, which is also request-based.

2.5.4 Selection drivers

The last section mentioned the influence of software tools (like Web Application Frameworks) on the productivity of development. However, there are multiple reasons why even the most productive framework is not always the first choice for a new application. Sometimes, other factors are so critical that productivity can no longer be a factor. Those criteria that influence the selection process are listed in the following. Any new Web project that is targeted with the selection of a WAF should take these factors into account.

PROJECT PARAMETERS The most important factor is the project itself. Either functional or non-functional requirements can push certain frameworks into the spotlight and disqualify others. A tool that has been the best choice for the last project can be unsuited for the next one. One way to start the evaluation process is to decide upon the type of WAF (see section 2.5.3) required: Request-based frameworks are said to be more scalable and "internet facing" than component based frameworks, best suited for a small number of clients or intranet applications ([63], [67]). Next, certain programming languages proposed by different WAFs might be no option due to performance reasons. Besides, WAFs are known to have certain "sweet spots" [62], that is, use cases they are well-suited for. If the application is one of them, the framework might be a good choice.

LEGACY SYSTEMS Often, legacy systems are another factor to deal with. For example, functional requirements might exist to connect the Web application to a legacy backend. In this case, choosing a WAF that is based on the same programming language could be a good approach [63], having a big influence on the possible framework alternatives. If there is a legacy Web application to deal with, depending on the WAF, the complexity of a conversion process might vary.

MATURITY Choosing a mature framework reduces the risk of a project. Otherwise, developers might have to deal with bugs and other shortcomings here. It can also be favorable to choose a framework that is based on approved standards. In this regard, Java's standardization approach in the Java Community Process (JCP) even allows choosing from different providers implementing the same API.

PROJECT HEALTH Applications planned to operate over multiple years should build on a WAF that is actively developed. Otherwise, the application might become obsolete over time. Frequent updates and support in mailing lists etc. can be valid indicators for the project health [63].

AVAILABILITY OF EXPERTISE Even if no new developers get hired for the application to develop, the availability of expertise still is an important factor. If the projects depends on the know-how of a few team-members, employee turnover can have bad impacts if no further expertise is available elsewhere. Against this background, job trends can also show up how a WAF develops in the market. If its popularity drops off, choosing the framework for a project expected to operate over multiple years might be risky.

LICENSING Besides license fees that may have to be paid (though unusual in the Java world), even the kind of open source license a WAF is published under may be important in special cases. Depending on the license, changes to the framework implementation have different effects on the resulting product. The difficulty regarding this point is that such modifications often cannot be foreseen.

Finally, managers should keep in mind that in the end, the developers are the ones using the respective WAF. Choosing one which they do not like will very likely decrease the team's productivity. Productivity as a main selection driver will be discussed in depth in chapter 3. This is also the place where the state-of-the-art regarding productivity measurement in web development is reviewed.

Chapter 3

Productivity in Software and Web Development

Productivity is an important factor in software development projects: Productivity gains result in shorter development times, they let the return on investment increase or they simply allow for a higher quality regarding the final product [5]. The other way round, a bad productivity may either result in a poor quality, in exceeded project schedules or even project cancellations – and a loss of money. Studies show that exceeded schedules and budgets are among the main reasons why development projects fail [30]. Irrespective of success or failure, the ability to meet schedule or budget targets is concerned "fair" or "poor" in one third to one half of all projects [30]. For the worldwide revenue only for enterprise software being estimated more than $230bn [61], productivity hence is a critical factor in software development. Accordingly, there is a huge interest to find those factors influencing it, from both academy and industry [22]. Studies on cost estimation already started in the early 1960's [66]. Regarding the last decade, they also increasingly address the area of web applications.

The two main fields of research are described in the following: These are productivity measurement (section 3.1) and the identification and estimation of factors affecting productivity (section 3.2). Section 3.3 describes the general relevance of web application frameworks in this context. Based on this theoretical analysis, the research questions for chapter 5 will be deduced (see section 3.4).

3.1 Measuring Productivity

For being so important, companies naturally have a high interest in controlling software development productivity. And, as DeMarco states: "You can't control what you can't measure" [25].

Software development is a process that encompasses multiple actions. The definition

of the productivity of a process is as follows:

$$\text{Productivity} = \frac{\text{Outputs produced by the process}}{\text{Inputs consumed by the process}} [11] \tag{3.1}$$

As the formula shows, productivity gains can be achieved by reducing the inputs, raising the outputs, or both [11].

Concerning software development processes, the consumed inputs comprise computers, supplies, other equipment, and labor [11]. For being the most expensive factor, labor, typically measured in man-months, has the highest impact here. Measuring the inputs is not trivial [11]. E.g., decisions need to be made which personnel to include into the calculations or which process phases to take into account. Nevertheless, companies can usually reach an agreement here. Such an agreement, if used in a consistent manner, allows for meaningful comparisons [11].

What makes productivity measurements really difficult is quantifying the outputs [21]. Many different ways for output measurement of software projects have been proposed over years. Such metrics can be divided into four different categories [21].

First, there is the large group of *physical size* metrics, measuring the Lines of Code (LoC) in the respective source code. Variations apply to the questions whether comments or reused code is counted or not. LoC metrics are widely used, though (or because of) being a quite basic approach [21]. Critics state that such physical size metrics tend to penalize high-level languages and have shown that well-formatted programs can be three times shorter than their original size (see references in [21]. A second group of metrics is based on the *functional size*, like function or use case points. Just like LoC, they are among the most used metrics [21]. *Design size* metrics measure the number of modules or classes. Finally, *value based* metrics measure added value or different aspects of output in multidimensional models. According to [21], value based metrics "show a good alignment with the idea of producing value and are gaining popularity today, but are more complex and still maturing". Some authors (see references in [21]) propose a completely different approach by measuring the value of a product from the view of its stakeholders. For all having different interests, their perception of productivity would also be different.

However, besides all these different approaches, there is evidence to suggest "that only very few organizations actually use them" [66]. There are various reasons for this, most of them related to the problem of how to define and measure output, intrinsic to the definition of productivity [21].

A universal model for measuring software productivity is not available [21]: Different companies and different projects underlie different circumstances. However, regarding single organizations, there often aren't enough resources available for the required measurement methods and tools. Hence, due to the lack of past project data, these companies cannot use meaningful metrics adequately [66]. If metrics are used nevertheless, this happens due to Gilb's law:

> *"Anything you need to quantify can be measured in some way that is superior to not measuring it at all."* [41]

However, authors also state that, depending on the indicators used, completely different conclusions about productivity can be drawn for the same project. They also mention the risk of "measuring dimensions that are not strongly linked to the phenomenon being observed" [21].

3.2 Productivity Factors

Another focal point of research in this area of software development is to find out the factors that determine productivity, and to which degree. This allows for a rating of the most important influences. Companies can adjust their efforts to increase productivity by making improvements in the corresponding areas. Corresponding surveys exist since the 1970's [57]. A good overview is provided by [22]. Though different factors and classifications exist, the estimation model discussed here is COCOMO II[4], for its cost drivers are the most widely selected in related studies [57].

> *"COCOMO II is an objective cost model for planning and executing software projects. [...] A cost model provides a framework for communicating business decisions among the stakeholders of a software effort."* [13]

COCOMO II provides support in financial decisions concerning software development projects, helps to set budgets and schedules or decides on tradeoffs among different influences like software costs or quality factors. To this, it is based on two underlying information models: A framework "for describing a software project, including models of the process, culture, stakeholders, methods, tools, teams, and the size/complexity of the software product" and a broad experience base of precedents [13]. Very important parameters in COCOMO II are the already mentioned *cost drivers*. These are characteristics of software development identified to affect the effort to complete a project. Depending on their impact on a particular project, they can be rated from "Extra Low" to "Extra High". Taking the ratio between the highest and the lowest rating value, the *productivity range* of a cost factor can be deduced. This is the factor to which a single cost driver can influence the costs of a project. E.g., by assuming all other factors are held constant, a productivity range of 1.25 for a particular cost driver means that depending on this cost driver, project costs can vary by 25 percent [11].

Figure 3.1 lists all productivity ranges from COCOMO II from the year 2000. For a detailed explanation of all the cost drivers, see [13].

The most significant influence is by far the management of people, their motivation and especially the individual developers that get deployed. It has to be noted that the impact of cost factors tends to change over time and that the depicted productivity ranges

[4]Project Site: `http://csse.usc.edu/csse/research/COCOMOII`

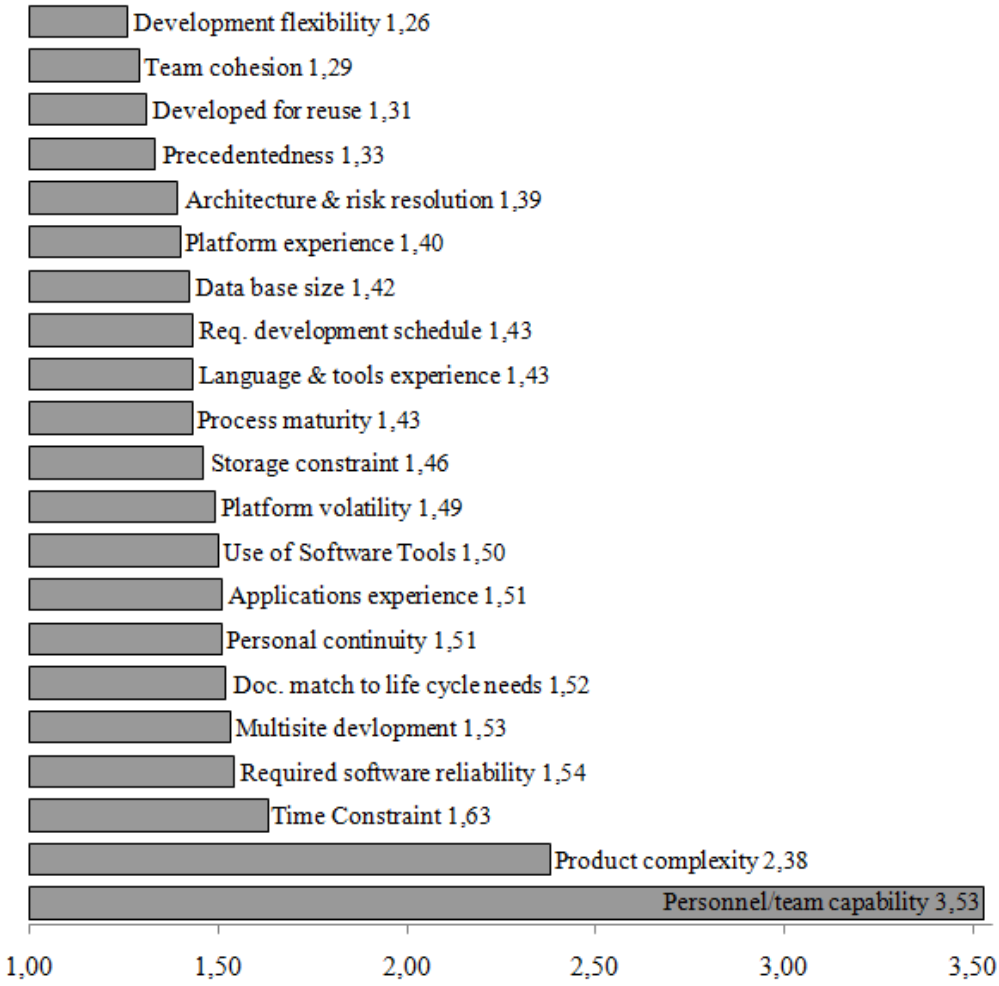

Figure 3.1: COCOMO II productivity ranges, 2000 [12]

go back to the year 2000. According to [57], the values are about to be updated soon. Moreover, as already mentioned for the metrics discussed in the last section, the precise values also depend on the respective environment. That is, organizations may need to calibrate them to improve estimation accuracies [57].

Figure 3.1 infers different strategies to improve productivity, the most promising ones being the following [11]:

- Get the best from people: As already stated above, the "personnel/team capability factor" by far has the highest productivity range. Comparing two developers, one can easily be three times more productive. However, companies will usually not need to pay him/her three times more. This means that organizations should always try to get the best from people. Besides staffing, improvements concerning facilities and management may also be worthwhile.

- Make steps more efficient: This is primary accomplished by using software tools and environments. An example for a more efficient development process is the use of a decent Integrated Development Environment (IDE) when none was used before. Another strategy in this category is the purchase of faster workstations.
- Eliminate steps: Again, this is mostly done by software tools. There exist different tools to automate repetitive and labor-intensive tasks. Automated testing or code generation approaches are all examples for this strategy.
- Eliminate rework: Eliminating the rework is one of the most compelling strategies, as rework usually encompasses around 30 % of the development process [11]. For improvements, companies can use front-end-aids (e.g., for traceability of requirements), adapt the process lifecycle or use rapid prototyping approaches. Other options relate to modern programming practices.
- Build simpler products: Companies should always aim for simpler products. This reduces the projects inherent complexity. Complexity has a productivity range of 2,38 and is the most significant cost driver after human capabilities. Rapid prototyping approaches can result in better architectures and less implementation efforts; thorough specifications may lead to less debugging and integration issues.
- Reuse components: Reusing components, libraries, code and specifications or designs is another key to productivity. This avoids recurring implementation efforts and thus, improves productivity.

3.3 The Influence of a Web Application Framework

The strategies for companies to improve productivity, as inferred in the last section, underline the major importance of software tools. Though these tools provide no support in personnel issues, they are relevant in any of the other strategies. The reason why software tools are nonetheless only ranked on ninth place in the list of productivity ranges is the amount of fixed product and team characteristics in this ranking: For example, the product complexity, the required reliability or the time constraints are either not management controllable or only to a small degree – they are part of the given project charter. Taking this group of cost drivers out of consideration, the use of software tools becomes one of the most important issues a manager has to take care of. Depending on the use of software tools, productivity may vary up to 50 %.

The productivity improvements gained through software tools can be of various types [14]: They can help to improve development processes by facilitating activities that haven't been practiced before. Development activities that existed before but were carried out without tools can be supported and sometimes even automated. Furthermore, they can improve quality and reduce rework in design and code. As more and more development tools become available, recent surveys [57] indicate that the impact of tools

is significantly increasing in the last 10 to 15 years. Consequentially, the most cited productivity factor in literature is "tools" (for a comparison, see [22]).

Obviously, in Web application development, the choice of software tools also includes the decision on an appropriate Web Application Framework. However, there is no precise estimation about their influence on productivity in literature. One method to determine such influences and get a rough impression is by *value chain analysis*. The value chain is a

> *"[...] method of understanding and controlling the costs involved in a wide variety of organizational enterprises. It identifies a canonical set of cost sources or value activities, representing the basic activities an organization can choose from to create added value for its products."* [11].

Particularly, by studying the value chain, enterprises can identify ways to save costs. Figure 3.2 shows a representative value chain of a software project.

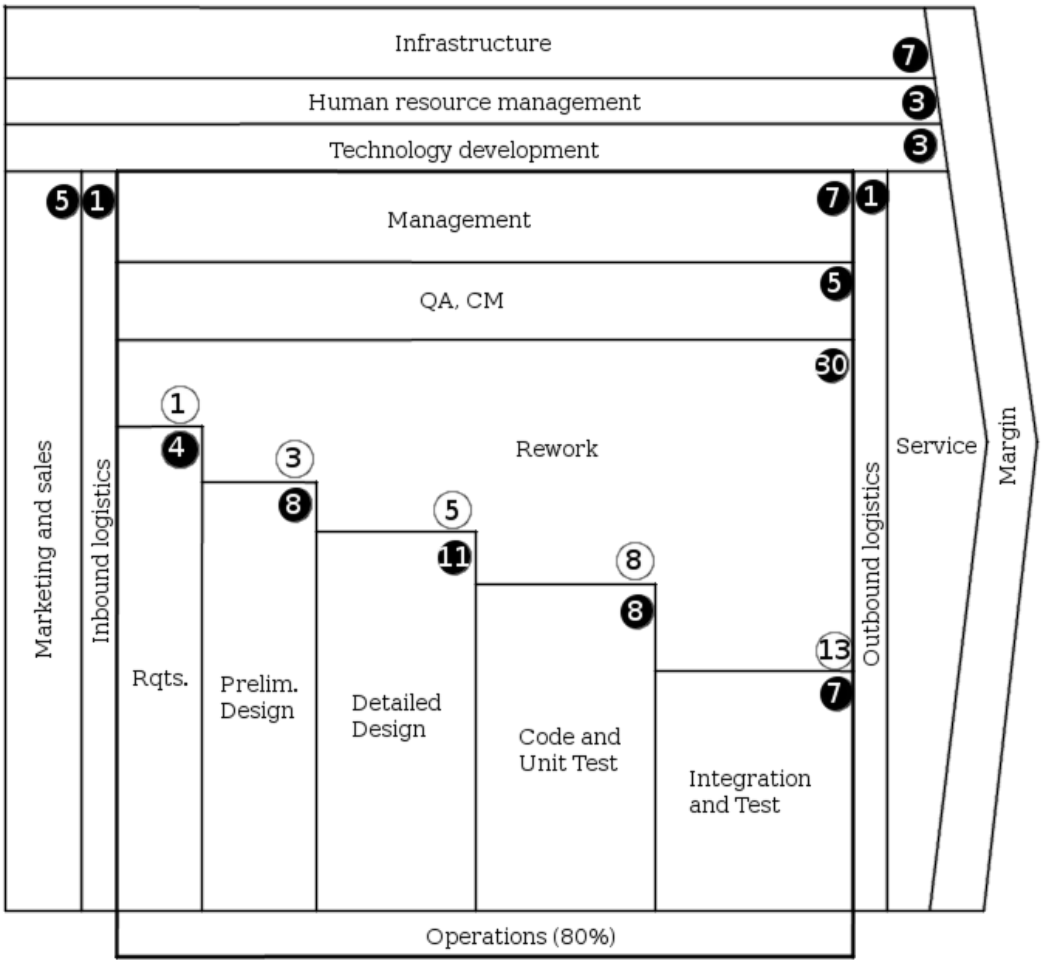

Figure 3.2: Exemplary Software Development Value Chain [11]

Note that some details have been omitted in the figure; for example, the service and margin components have not been assigned numbers. The service activity, above all including maintenance and evolution of software, typically causes 70 % of lifecycle costs [11]. However, according to [11], the value chain for this specific phase "is not markedly different" from the general distribution of costs in figure 3.2, therefore enables no additional exploration.

The most important thing to note is that the operations phase covers 80 % of the overall development outlay. This is the area where software developers are directly involved and it therefore is the key area to significant productivity improvements in software engineering. The block breaks up into Management (7 %), quality assurance and configuration management (5 %) and the different development phases. Including rework tasks, coding, integration and testing altogether make up 36 % which is 45 % regarding the sole operation costs. In Web Engineering, these are the areas that are usually accompanied by a WAF.

Another thing to note is that the depicted value chain goes back to the year of 1987. Since that time, many changes occurred regarding software engineering practices: "Software Processes have transformed from code-and-fix and waterfall-like to spiral, iterative, and more agile approaches to developing software" [57]. For such iterative and agile processes are known to have different perceptions, one of them for example to value working code higher than extensive documentation [44], the proportion of work influenced by a WAF is probably even higher.

Furthermore, the inversion of control principle inherent to WAFs (see 2.4.1) may additional increase their importance in the value chain. Frameworks influenced by this architectural style tend to impose a specific design to the resulting application, because they clearly define the points where control is handed over to the developer. Usually these designs align to the best practices in the respecting areas, like the MVC pattern for web applications (see 2.3). The usage of a WAF therefore significantly simplifies the design of the application, usually accounting up to 27 % of the value chain costs.

The usage of frameworks also allows for (third party) code reuse. The reuse of software artifacts avoids unnecessary work. According to Boehm in [10], work avoidance can save 47 % over the "normal improvements accrued via a business-as-usual approach". However, he stresses that there are several pitfalls to avoid when improving productivity this way. But regarding frameworks based on inversion of control, like WAFs, this reuse is automated to a large degree. This is another indicator for the influence of WAFs on development productivity. Other features provided by them were already mentioned in the last chapter, like scaffolding and template approaches, and are about to be discussed in detail in the following.

There also exist special cost estimation models for web applications, as for example in [66]. The reason for this is the sum of special characteristics inherent to web application, as already discussed in section 2.2.3.

However, in some situations software tools can also decrease a project's productivity.

This may happen because they increase the effort on specific activities or even introduce new activities into the process [14]. One important factor to consider here is the tool's learning curve.

The learning curve specifies how much time needs to be invested into learning the tool before becoming productive with it. A WAF's learning curve highly depends on the know-how in the team. Depending on the background of the respective developers, it might be steeper or smaller. [5] Nevertheless, this is also a property of the framework but difficult to objectify. A decent documentation eases the start with the WAF. However, published books could even indicate that the respective framework is too difficult to learn without them. The longer development takes or the more projects are going to be build with the respective WAF, the lesser is the influence of the learning curve.

3.4 Research Questions

This diploma thesis studies the productivity gained by web application frameworks. In the context of upcoming web application development projects, it aims at simplifying the process of selecting the appropriate WAF when productivity is a criterion – which does not always have to be the case, as section 2.5.4 discussed. However, against the background of the current state-of-the-art in software productivity research, the question is coming up, what the best methodology on this would be. As the past sections showed, the research in the field focuses on two main areas: Productivity measurement and productivity factors.

Productivity measurements are mostly driven by an ex post view and require a project database for analysis. In the literature, there is one example ([66]) where Ruhe, Jefferey and Wieczorek studied cost estimation for Web applications. The data set used for the study consisted of twelve "typical" web applications, created between 1998 and 2002. For analysis, they adapted the so-called COBRA method, which, like COCOMO, is also based on cost factors. However, they clearly state that there is no experience on potential cost factors for the domain of Web applications. Hence, expert interviews where used as estimation. To measure the size of the respective Web applications, function points were counted. This resulted in a list of "potential" cost factors of web development. However, due to the small-sized project database, the significance of these results can be challenged. Furthermore, the influence of tools was not taken into account.

To actually measure the productivity of different WAFs this way, a much bigger web project database is required but does not exist. But even with an existing project database, the problem of how to measure output (see section 3.1) would still exist.

The fact that the influence of development tools was not taken into account reveals that an analysis of productivity factors regarding Web Application Frameworks is re-

[5] Whether a steeper or smaller learning curve is better depends on where time and outcome are mapped on in the coordinate plane. Regarding other measurements, time is usually mapped on the abscissa. This would mean that a steeper learning curve is better. However, developers and even scientists most often speak of steep learning curves being bad, so they obviously map the outcome on the abscissa.

quired. As [73] concludes:

> *"No comprehensive list of functional requirements of Web frameworks has been found in the literature."*

However, the authors themselves do also only provide a list of seven topics, admitted to be "slightly generalizing". Hence, the features provided by WAFs haven't been studied in its completeness yet, especially not those that account for their productivity. But without obtaining this knowledge, a decent analysis and comparison of WAFs to finally choose among them will not be possible:

> *"Because we don't know how to analyze a tool's impact on specific projects, we generally adopt them based on an intuitive understanding of their expected impact. In many cases, the actual results of this practice are disappointing."* [14]

Though this refers to software tools in general, it is also valid for WAFs. Already in May 1992, the theme of IEEE Software's special issue on Tools Assessment was:

> *"The industry has made great strides in making more development tools available. It's now time to find ways to consistently, objectively evaluate a tool's utility and appropriateness."*

A catalog of technical criteria that form a WAF's productivity would allow such objective evaluation.

Research question: Which technical criteria make up the productivity of a Web Application Framework?

In [46], the authors have created a comparison model for Agile Web Frameworks. To this, they defined a set of aspects that "summarize general features and issues found in web application development". These general aspects were also used as a basis for this work. However, the resulting comparison model was much too coarse-grained and there was no clear focus on productivity.

To make a detailed analysis possible, the following examination will only comprise the Java market and confine on server-side aspects. This way, according to the taxonomy introduced in section 2.5.3, the analysis will be both valid for request- and component-based frameworks. The final catalog divides into six major parts, namely (1.-3.) the support regarding the three components in the MVC pattern (see section 2.3), (4.) how development speed can be fostered irrespective of a particular artifact under development, (5.) build and integration support and finally (6.) testing. This classification was both derived from [46] and the author's experience gained by analyzing the different WAFs presented shortly in the next section: It reflects the core areas where modern web

development tools set their focus regarding productivity enhancements. Whenever different ways exist to fulfill a certain task, the pro's and con's of each approach, especially the required input, will be examined and their consequences on the application's quality are discussed.

As stated in [14], a "tool's impact is not solely governed by its inherent properties, but also by the characteristics of the adopting project." That is, the actual impact of a particular productivity factor depends on the respective project. This is a general issue inherent to all surveys on productivity factors though (see section 3.2). Nevertheless, by reading chapter 5, one should get an impression on the influence of each criterion. Furthermore, as a quick overview, Appendix A classifies the criteria and their influence on productivity ranging from `LOW` to `HIGH`. This classification is based on the author's impression gained throughout the work. A more objective measurement would again require some kind of project database.

Finally, as [14] emphasizes, "to make effective use of specific tools you should first understand how a tool will affect [...] critical variables in your project." Regarding the development with Web application frameworks, the following chapters of this master thesis provide a good starting point to answer this question. The current state-of-the-art in rapid web development will become clear and a forecast on future developments in the field is given. Knowing such trends can be helpful for managers to select future-proof solutions in their web engineering projects but may also be interesting for WAF developers to decide on their next moves in implementation.

Chapter 4

Examined Web Frameworks

In this chapter, the three Web Application Frameworks analyzed in the course of this work, namely JBoss Seam, Grails and Spring Roo, are described. A deeper insight would go beyond the scope of this work though. However, additional information about each framework is referenced in the respective sections. By describing (1) the history of the framework and the motivation behind it, (2) its most prominent features and (3) special approaches to productivity, the spirit behind each implementation should become clear. The chapter also provides a first impression of the current state-of-the-art in rapid Web development.

4.1 JBoss Seam

JBoss Seam[6] is a framework for building Web applications using the Java EE platform [56]. The name alludes to the term of a *seamless integration* and was originally related to the fact that the framework fitted JSF and EJB, two major specification in Java EE, closer together [4]. Today, Seam unifies many parts of Java EE and aims at making other proven libraries more accessible. Many developers choose the framework to deal with shortcomings of the JSF specification or because they recognized Object-Relational Mapping (ORM) related problems in alternative WAFs. This way, people around Seam had a big influence on the latest JSRs for Java EE 6, especially JSF 2, CDI and Bean Validation: Many were part of the respective expert groups or even led them.

Besides this heavy scope on Enterprise Java, Seam also considers many concepts of rapid Web development and ships with a rich tool set fostering agile development.

[6] Project Site: `http://www.seamframework.org`

4.1.1 Motivation & History

Seam was founded in 2005 and is actively developed by JBoss, a division of Red Hat. The former Project Lead was Gavin King, already known as the initiator of the ORM framework Hibernate. The reasons for creating a new WAF were therefore closely connected to the big success of Hibernate.

King recognized that many developers were using Hibernate improperly. Relying on a stateless design, propagated particularly by the Spring Framework, they could not take advantage of the extended persistence context [4]. This often resulted in detached objects and Hibernate's `LazyInitializationException` or in many unnecessary database calls, the number one reason for performance problems in today's software applications [42]. Other developers tried to solve this problem by scoping the persistence context to the user's session, held it open for too long and ended up in cache and memory leaks. King's idea was to permit a stateful design in which the persistence context could be managed automatically, extended to survive multiple requests, by better integrating JSF and EJB 3. Today, this continuity between user interactions still is one of Seam's core competencies.

Seam is licensed under the GNU Lesser General Public License (LGPL). At the time of writing, the latest final release of Seam was 2.2.0.GA which is also the version under examination in the course of this work. Seam 3 is under development and is proposed to implement all outstanding Java EE 6 features.

4.1.2 Prominent Features

Seam provides many features to ease Web application development. The ones described in the following are for the most part unique to the platform and fundamental for developing with Seam. Chapter 5 will refer to many other features the framework provides. A complete overview of Seam can be found in [52].

JSF enhancements

The previous versions of JSF 2 contained some serious shortcomings [4].

The lack of a front controller made many use cases very difficult to implement. *Initial requests* were not able to do much more than serving a page. Invoking action methods, required to e.g. serve RESTful URLs, making security checks or redirect to a different (error) page, was not possible without falling back to a custom `PhaseListener`. This introduced lots of boilerplate code into the application logic. Seam provided an approved and configurable, yet transparent `PhaseListener` to solve the problem.

Postbacks, the other request type in JSF, were also problematic. Since posting data can easily interfere with the browser's back button and its ability to set bookmarks, the Redirect-After-Post Pattern has evolved. Unfortunately, JSF 1.2 offered no way to store request data anywhere else than in the user's session. This resulted in performance problems and multi-window complications. Though also based on traditional Java Servlets,

Seam was able to provide additional, declaratively manageable scopes to the developer. By controlling the session context in a fine-grained manner, conversations and even business process contexts became possible.

Another problem of the old JSF was to connect it with EJB. Though both technologies where important specifications in Java EE 5, JSF required the additional layer of a backing bean to finally access an EJB. This resulted in a tighter coupling between the view and the business layer. In Seam, Enterprise Java Beans can be used just like normal baking beans and can be called from the view by using the enhanced EL. A concrete example how Seam improved applications in this regard is provided by [7].

The aforementioned information may appear redundant because in the meantime, all these problems were also solved by JSF 2. However, Seam's success bases particularly on these enhancements which were probably the main reason for companies to choose this framework for their developments. Moreover, they were obviously of big influence on the latest JSF specification.

Bijection

The concept of *Dependency Injection* was already explained in section 2.4.1 on page 14 as a way to keep components loosely coupled in a POJO-based programming model. Seam extends this mechanism in two ways [4]:

First of all, dependency injection is dynamic in Seam. Instead of injecting an object only once during initialization, Seam repeats this step every time a method is called for the respective component.[7] Otherwise, injecting a component instance of a narrower scope into a component instance stored in a wider scope could easily result in a `NullPointerException`, because the lifetime of the used component has already come to an end.

Besides moving away from static dependency injection, Seam also adds the concept of *Outjection*. Outjection means that a component itself can define new context variables of a specified scope. In a second step, these variables can then be used by other components, for example by using dependency injection again. This way, outjection becomes an easy way to propagate state held by a component into the Seam container.

The process of injection and outjection is called *Bijection*. Bijection proceeds in four steps [52]:

1. Dependency injection for fields annotated with `@In`
2. Invocation of the called method defined by the component
3. Outjection for fields annotated with `@Out`
4. Disinjection

[7] Exceptions occur for *reentrant* method calls.

To implement this mechanism, Seam uses a *bijection interceptor* [52]. After the invoked method has finished and outjection was performed, *disinjection* assigns null values to all formerly injected fields to remove all data that could not be kept up to date anyway while the component is idle. This prevents memory leaks.

There are additional variants of `@In` and `@Out` to mark fields for bijection. It is also possible to configure bijection using XML.

4.1.3 Special Approaches to Productivity

Like any other framework examined in this master thesis, Seam tries to foster an agile development. The different approaches hereunto are partly presented in this section and will be discussed in depth in Chapter 5.

The project generator

The project generator *seam-gen* can be used to create executable Web applications, capable to *create*, *read*, *update* and *delete* (CRUD) database records, to find and filter, sort and paginate them. A skeleton project created this way is fully configured: It is ready for use in the most popular IDEs, provides ant-scripts[8] to deploy on JBoss AS or Glassfish, and supports hot deployment of static resources, views and Java Beans. Different development environments are created, just like the support for AJAX or a component debug page for development.

To benefit from all these features, which are a great relief for everyone starting with a new framework, the user has to answer a small questionnaire about the project to be created in the beginning. This information includes database settings and basic project facts like name and package format (WAR or EAR).

The user can also specify whether to follow a *bottom-up* or *top-down* approach on the persistence layer. Bottom-up in this case means to create JPA-entity classes by examining an already existing database schema. In the background, seam-gen uses Hibernate's reverse engineering tool[9] to accomplish this goal. Following the top-down approach is the common way of implementing entity classes which define the database schema [52].

After this scaffolding process, seam-gen may still be used to update the project, create Seam components or generate views for added entity classes.

JBoss Tools

For the most part, the JBoss tools[9] are designed for use in all kind of projects, not in a Seam-specific way. One such example is the already mentioned reverse engineering tool. Others support the creation of JSF-views in a WYSIWYG manner or provide code-completion in IDEs for special file types. Some tools come as an IDE plug-in, others are

[8] Project Site: `http://ant.apache.org`

[9] Project Site: `http://www.hibernate.org/subprojects/tools.html`

designed to be used in the console only. Even when not using a single JBoss technology at all, the tools can be of great help for a Java developer.

However, some of the tools are specific to JBoss Seam. They comprise wizards to create new Seam projects or add specific seam components, editors for visual support or content assist when defining components. A complete overview is provided by [33].

Query & Home Component

Bundled with Seam comes the so called *Seam Application Framework*, basically a set of classes to support the developer in building components that need to perform database operations. Managing persistent entities is not always trivial and, when done wrong, may result in serious performance problems. The Seam Application Framework aims at avoiding these problems and additionally allows for a quick development of such use cases.

Among the most important classes are two kinds of *persistence controllers*. A persistence controller implements the mediator design pattern by encapsulating the respective persistence manager and a page controller in a single component [4]. This approach has a major advantage: The persistence manager as a replacement for DAOs used in other WAFs is able to perform CRUD operations and manage the results. By scoping the persistence controller to the appropriate context, the developer does no longer have to fear detached objects or performance problems. However, the persistence manager is unaware of all the things happening on the Web layer – this is where the page controller comes into play. By combining both in a single unit, database operations can be easily accomplished in a both stateful and performant manner. All this should become clearer by describing the two major persistence controller templates: `Home` and `Query`.

The `Home` Component manages a single entity instance. For initialization, the developer needs to provide the entity class of the instance to be managed and the persistence provider to be used, either JPA or Hibernate. After this, `Home` allows for a stateful CRUD of single persistent entities.

In comparison, `Query` can manage complete result set listings. By providing a query, the developer can easily truncate and paginate the results, sort them according to the user's preferences or apply additional restrictions. By storing the `Query` component in the right context, the developer does not have to fear issuing unnecessary database calls or running out of memory, even if the use case involves multiple HTTP requests.

4.2 Grails

Grails[10] was invented with the main intent to be the "better, easier way to build Web applications on the Java platform" [71]. It is highly influenced by Ruby on Rails[11] (in the beginning, the original name of the framework has even been *Groovy on Rails*[12]). As such, principles from agile and rapid development are deeply integrated into the framework [71].

Grails applications are usually written in *Groovy*, a dynamically typed language for the JVM. Though a different programming language, developers can benefit from all the classes and methods provided by the Java Development Kit (JDK) as usual – even circular dependencies between Java and Groovy code are no problem.

Another point that sets Grails apart from other WAFs in the Java environment is that it is indeed grounded on commonly used technologies like Hibernate or Spring, but sets an abstraction layer on top of them to shield the developer from their complexity [64].

Since the programming model differs widely from the already discussed JSF or Seam, a code example is provided in section 4.2.3 to get an impression on how a Web application is designed using Grails.

4.2.1 Motivation & History

The history of Grails is closely connected with the history of Groovy. Both were triggered by developments outside the Java platform.

Groovy was invented in 2003 by James Strachan. Strachan was a Java developer, but when comparing Java with other programming languages like *Python*[13], he realized his language was lacking many useful features, especially dynamic behavior. For Java still being a robust and well-supported platform, he decided to port these features to the JVM. The principle of a Java-friendly but feature-rich programming language still is what drives Groovy development today [53].

In 2004, Groovy was standardized by JSR 241 as the second standard language for the JVM. According to Sun, "having additional interesting languages for the Java platform seems like a Good [sic] Thing"[14], that is, they see no rival but a companion for Java in Groovy [53].

In the Same year, *Ruby on Rails* was released. Many developers were attracted by the quick and easy way Rails provided to create data-driven Web applications. At that time, Graeme Rocher was already implementing MVC-Controllers in Groovy, making use of a self-adapted version of WebWork[15]. Recognizing the success of Ruby on Rails and other WAFs built on dynamic languages, Rocher and Guilaume Laforge, specification

[10] Project Site: `http://www.grails.org`
[11] Project Site: `http://rubyonrails.org`
[12] `http://marc.info/?l=groovy-dev&m=114373309105440&w=1`
[13] Project Site: `http://www.python.org`
[14] Sun's comment in Vote Log for JSR 241
[15] Project Site: `http://www.opensymphony.com/webwork`

lead for JSR 241 and project lead for Groovy since that time, decided in 2005 to create a WAF based on Groovy. Since November 2008, Grails is part of the SpringSource[3] portfolio [64].

The current stable version of Grails is 1.3.6. It is published under the *Apache License*.

4.2.2 Prominent Features

In comparison to other Java Web Frameworks, Grails most prominent feature is of course Groovy. For Groovy being a complete and self-contained programming language, discussing all its features will not be possible here. By explaining Groovy's basics and providing some code examples, this should however not influence the further understanding of Grails.

Another concept that sets Grails apart is its plug-in mechanism which will be discussed in the second part of this section.

Groovy

Groovy[16] is a dynamic programming language allowing both static and dynamic typing which can be mixed arbitrarily. In contrast to other dynamic languages, Groovy code is not interpreted but transformed into bytecode [53]. This point results in Groovy's core strength which makes it so compelling to Java developers: Java code can call Groovy code without even being aware of the fact that it was written in a different language. The same holds true for Groovy code accessing objects written in Java.

This does not mean that any code fragment taken form Java is also valid Groovy (though this applies for many Java classes). For example, Groovy has limited support for Java's classic `for`-loop and implements no equivalent for inner or anonymous classes [53]. But aside from these constructs, Java and Groovy are very similar, which makes Groovy easy to learn with a background in Java [71].

Groovy is more expressive than Java and often better to read [53]. First of all, Groovy tries to fix many problems that became obvious in 15 years of Java programming. These concepts will be discussed shortly. Furthermore, Groovy aims at providing programming concepts to the Java platform already proven powerful in other languages. Among these are Closures and Metaprogramming. For being very important concepts also in Grails, both will be discussed separately. To better understand all these features, an example is provided in each of the following sections. These examples are *Groovy Scripts*, that is, they are executable without any further ado.

All in all, Groovy programs usually contain only 40-60 % of the LoC an equivalent Java program would require [64].

[16] Project Site: `http://groovy.codehaus.org`

SYNTAX Groovy simplifies Java's syntax in many concerns. Semicolons at the end of lines are not required; parentheses can often be omitted (though it is recommended doing so only in the simplest cases [53]). Where Java only imports the `java.lang` package implicitly, Groovy does so for many more, including `java.io`, `java.util` and `java.math`. Furthermore, Groovy stops the coexistence of primitive types and objects known from Java: In Groovy, everything is an object. These objects also have a different scoping behavior: If no scope is specified, methods are concerned `public` and fields turn into properties, that is, getters and setters become available during runtime and accessing the field directly by its name results in invoking the respective Getter/Setter [53]. Java's *default scope* has been completely removed. Another important thing to note is that Groovy uses the `BigDecimal` class for its floating-point arithmetic instead of `double` or `float`. Though inferior in performance, this follows Groovy's principle of least surprise: For example, `0.1D * 3` is `0.3` in Groovy but `0.300...04` in Java [53]. Together with its wide simplification of the `BigDecimal` API, this makes Groovy an interesting option for financial calculations.

Listing 4.1: Accessing properties and the usage of Big Decimal in Groovy

```
1 class Person {
2     String name
3 }
4 def person = new Person()
5 person.name = 'John Doe'
6 assert person.getName() == 'John Doe'

8 assert 2.0 instanceof BigDecimal
9 assert 2.0 / 0.5 == 4
```

NEW OPERATORS Groovy provides many new operators to the developer. The spread-dot operator (`*.`) can be used to invoke a method on every item in an aggregated object and replace the item by the method's return value. Using the safe navigation operator (`?.`), the developer no longer has to fear an object he is calling a method from may be null. In this case, the whole expression returns `null`, but throws no `NullPointerException`. Chaining this operator is possible to an arbitrary depth. The Elvis operator (`?:`) is a simplification in cases where using Java's ternary operator becomes quite verbose to either assign a value to a variable or – in case this value is null – set a default value. One very important thing to note for Java developers is that Groovy uses == for comparisons on equality and the `is` operator to check for identity. Groovy even adds an operator to easier compare two values (<=>) according to Java's `Comparable` approach. Furthermore, the `as` operator allows for easier type casting.

There are lots of other operators for special use cases such as regular expression matching. Further information is provided by the Groovy User Guide[17].

[17] `http://groovy.codehaus.org/User+Guide`

Listing 4.2: Additional Operators in Groovy

```
1 def list = [1,2,3]
2 assert list*.multiply(2) == [2,4,6]

4 Date date = null
5 assert date?.time == null

7 String oldName = null
8 String newName = oldName ?: 'John Doe'
9 assert newName == 'John Doe'

11 assert 'Amy' <=> 'Zue' == -1

13 assert 12345 as String == '12345'
```

LISTS, MAPS AND RANGES Collections are a very important concept for every programmer. In Java however, collections aren't much more than a set of APIs. Groovy makes lists and maps first class citizens of the language and even add the concept of ranges for easier creating enumerations of arbitrary lengths. Parsing or constructing a collection is less verbose this way. Making use of operator overloading, the same holds true for the way new elements are added to lists and maps. Requesting a value from a list is supported in an array-style notation. Even addressing an element starting from the end of the index is possible.

However, what makes lists and maps even more powerful are the methods that Groovy adds to their APIs, profiting from the two concepts defined next.

Listing 4.3: Working with Collections in Groovy

```
1 List list = []
2 for(i in 1..100){
3     list << i
4 }
5 assert list.size() == 100

7 list = [1,2,3,4,5]
8 assert list[0] == 1
9 list = list[0..2]
10 assert list[-1] == 3
```

CLOSURES Closures are a powerful concept in dynamic languages. Though appearing complex in the beginning, they quickly become invaluable to the programmer after understanding their usage [71].

Closures are reusable code fragments and behave like anonymous functions [71]. One might also think of closures as a method without a name. However, unlike methods, the closures arguments are declared within the curly braces that also encompass

the closure's body. The general syntax of a closure in Groovy is `<arguments> -> <body>` . In case no arguments are specified, the default argument `it` can be used within the body and the arrow (`->`) has to be dropped. The body is a sequence of instructions, where the last instruction defines the body's return value. Using the `return` keyword to this is not compulsory; the same rule applies for any method written in Groovy. Defining a closure can either be done explicitly or inline.

The *Groovy JDK*[18], which defines the methods added by Groovy to the normal JDK, heavily exploits the concept of Closures. This is another reason which makes Collections so powerful in Groovy.

Listing 4.4: Working with Closures in Groovy

```
1 Closure sqr = {a -> a * a}
2 assert (1..3).collect(sqr) == [1,4,9]
3 assert sqr(4) == 16

5 def sum = 0
6 (1..100).each{sum += it}
7 assert sum == 5050
```

CATEGORIES *Categories* are Groovy's way to extend the behavior of already existing classes and are one part of Groovy's Meta Object Protocol (MOP) [23]. "Extending" in this sense is different from the concept of inheritance known from object oriented programming: Groovy makes it possible to add methods to classes even declared as `final`. E.g., the Groovy JDK adds many additional methods for Java's `java.lang.String` API. Likewise it does for dozens of other classes in the JDK. The following code listing is an example of a user defined category for the already mentioned `java.lang.String` class. Such methods are defined static and programmers need to enclose code sections where they wish to use the newly defined method with the `use` construct (this is not required for the categories defined in the Groovy JDK though).

Listing 4.5: A user-defined method (Category) for `java.lang.String`

```
1 class StringUtilCategory {
2     static def firstCharacter(String self){
3         return self?.substring(0,1)
4     }
5 }

7 use(StringUtilCategory){
8     assert "John Doe".firstCharacter() == 'J'
9 }
```

[18] `http://groovy.codehaus.org/groovy-jdk`

METACLASSES Metaclasses are another important concept in the Meta Object Protocol. In contrast to categories, they allow to dynamically change an object's behavior. This feature is also known as *Duck Typing* in dynamic languages ("if it walks like a duck and talks like a duck, its probably a duck") [71]. In Groovy, duck typing becomes possible in the following way: Every Groovy-object implicitly implements the `groovy.lang.GroovyObject` interface.[19] The interface defines a couple of methods, the most important ones here are `invokeMethod()` and `setMetaClass()`. Every object has a metaclass set by default. The metaclass maintains all the metadata of an object, i.e. constructors, fields, operators, properties, static methods and instance methods. By default, `invokeMethod()` only forwards requests the object's metaclass [53]. That is, by setting a different metaclass for an object, you may alter the object's behavior at runtime. Another option would be to overwrite `invokeMethod()`. This way, the programmer becomes able to *pretend* behavior [23]: If the metaclass does not implement a method that the object calls on it, a *MissingMethodException* will be thrown. `invokeMethod()` may catch this exception and dynamically generate the appropriate behavior. For example, Grails Object Relational Mapping (GORM) makes heavy use of this option by providing *dynamic finders* [53]. This way the developer may call methods like `Person.findAllByLastnameLike("Doe")` though he never wrote the respective method: Dynamic finders generate the appropriate query automatically, provided `Person` implements a property called `lastname`.

Plug-ins

Plug-ins are modules that bundle functionality. This functionality often isn't specific to one single application and therefore can be shared between projects. Using plug-ins means reusing code. Code reuse has many advantages: First, the developer saves the time of developing an own solution. Second, the overall quality of the application under development increases, for the reused code is often well tested, robust and mature [71].

Grails was built with plug-ability in mind since its beginning [20]. In a way, Grails is not much more than a runtime-plug-in loading and configuring other plug-ins. Accordingly, Grails' architecture consists of the Grails core and a plug-in system connected to it [71]. Plug-ins themselves divide into *core plug-ins* shipped with Grails and *user plug-ins* provided by the community. One example for a core plug-in is the already mentioned GORM. User plug-ins can be obtained from Grails' plug-in repository. Currently, more than 400 user plug-ins exist. For example, there is support for mail, full-text searching the database or AJAX. Obviously, since these are no official releases, quality differs widely. Some serve the same purpose or are out of date and can't be used with newer Grails versions. Contrariwise, others reveal so useful and mature that they become part of the core plug-ins. At the Grails *Plugin Portal*[20] one can get an overview about the

[19] This is the same interface a Java class needs to implement so that intances are treated like normal Groovy-objects.

[20] `http://grails.org/plugin/home`

experiences developers made with a specific plug-in. Before developing a feature requirement that is not application-specific, a project team should always check whether there is already a plug-in provided for this purpose.

The modularization implied by using plug-ins may lead to an improved architecture on the respective application. This is the reason why it sometimes makes sense to implement parts of the applications functionality as a plug-in, even though it is not going to be available to anyone else. A plug-in in Grails is a Grails application; therefore, development is quite natural for developers familiar with the framework. Besides finding and installing plug-ins, the Grails command line tool (see section 4.2.4) also supports the creation of new plug-ins. However, new plug-ins should stick to the principle of convention over configuration and avoid being intrusive to the developer. Plug-ins may even base on already existing plug-ins [64].

4.2.3 Programming Model of Grails

As already stated, Grails applications differ widely from component-oriented JSF applications. Therefore a short example will be depicted in this section to get an impression on the programming model in Grails. This should also lead to a better understanding of Spring Roo (see section 4.3 on page 51), since designing a Web application there is pretty similar to Grails. The files shown in the following do not make up a fully executable application of course, but by using one of the tools explained in section 4.2.4, the remaining work basically comes down to configuring the data source.

As the name already says, a *domain class* describes a part of the application's domain. The domain in the following is quite easy: The database stores a number of messages and should be able to present a single message to the user. By providing the id of the message, the user itself can decide which message to display.

Listing 4.6: The Message domain class (`Message.groovy`)

```
1 package messenger

3 class Message {

5     String author
6     String message

8     static constraints = {
9         author(blank:false)
10        message(blank:false,maxSize:255)
11    }
12 }
```

As shown in listing 4.6, the domain class contains fields for saving the author of the message and the message itself. Behind the scenes, Grails will also add an *id field* and

a *version field* for optimistic locking. Both the author and the message field must not be null or empty ("blank") and the maximal length of each message is 255 characters. The class represents a valid Grails Object Relational Mapping (GORM) file. GORM is a *Domain Specific Language (DSL)* Grails uses as its ORM-Layer. In contrast to general purpose languages like Groovy or Java, a DSL is "a computer programming language of limited expressiveness focused on a particular domain" [39]. The domain in this case is ORM. Groovy is very powerful in creating domain specific languages.

Grails will automatically generate multiple methods for the domain class, so that everything is already there to be able to retrieve messages from the database. The respective logic will be called by a *Controller*. A controller in Grails performs the same tasks as the controller known form the MVC pattern (see section 2.3). Here, the controller has to read a message id from the user request, get the respective message from the database and return a view back to the user presenting the information.

Listing 4.7: The Message controller (`MessageController.groovy`)

```
1 package messenger

3 class MessageController {

5     def show = {
6         def msg = Message.get(params.id)
7         if (!msg) {
8             flash.message = "Message ${params.id} not found"
9             redirect(uri:'/index.gsp')
10        }
11        else {
12            [msg: msg]
13        }
14    }
15 }
```

The method `get()` in line 6 of listing 4.7 is automatically generated by Grails for every domain class. `params` is a map that stores request parameters by their corresponding name. In this case, the id of the message the user requests is needed. If a message was found, a request-scoped map containing the entity is returned and the default view is displayed, which in this case is "list" because this is the name of the called closure. Otherwise the user is redirected to another page, displaying the appropriate error message. Since request-scoped data gets lost on a redirect, the message needs to be stored in the flash scope.

If the user requests the URL `app/message/list/{id}`, the `list()` method in the MessageController gets called and the value of `{id}` will be bound to `params.id` (assuming that `app` is the name of the Web application under construction).

All that remains is displaying the message to the user. By default, Grails uses Groovy Server Pages (GSP) as its view technology, which is very similar to JSP [71]:

Listing 4.8: Displaying a single message (`list.gsp`)

```
1 <%@ page import="messenger.Message" %>
2 <html>
3 <body>
4    <g:if test="${flash.message}">
5       <div class="message">${flash.message}</div>
6    </g:if>
7    ID: ${fieldValue(bean: msg, field: "id")}<br>
8    Author: ${fieldValue(bean: msg, field: "author")}<br>
9    Message: ${fieldValue(bean: msg, field: "message")}<br>
10 </body>
11 </html>
```

Line 5 in listing 4.8 describes a way to read-out flash-scoped data as created by the controller. To access data stored in the different scopes, the view has to use the appropriate keys (namely `message` and `messageInstance` in this case).

4.2.4 Special Approaches to Productivity

Grails was designed to allow for a rapid development of Web applications. Groovy and Grails plug-in support, two major approaches into this direction, were already discussed in section 4.2.2.

Just like Seam, Grails offers a project generator and support for developing Grails applications inside an IDE. For the concepts are pretty similar to the ones described in section 4.1.3, the following information is a bit shorter. This does by no way mean Grails support in this regard is less powerful or provides less functionality.

Grails' command line tool provides a way to generate fully configured CRUD applications. Using the tool, new domain classes, controllers or views can be created or generated. There are also commands to create and run test case or look for or install new plug-ins. The console also allows for generation of new *gant* scripts. Gant[21] is a build tool for Groovy that encloses Apache Ant [64].

Different IDEs offer plug-ins for Groovy and Grails, including NetBeans,[22] IntelliJ IDEA[23] and Eclipse.[24] The SpringSource Tool Suite[25], discussed in section 4.3.3, probably offers the best support.

4.3 Spring Roo

Spring Roo[26] takes a completely different approach than the two WAFs discussed before. According to the project's mission statement, Roo wants to "fundamentally and sustain-

[21] Project Site: `http://gant.codehaus.org`
[22] Project Site: `http://netbeans.org`
[23] Project Site: `http://www.jetbrains.com/idea`
[24] Project Site: `http://www.eclipse.org`
[25] Project Site: `http://www.springsource.com/developer/sts`
[26] Project Site: `http://www.springsource.org/roo`

ably improve Java developer productivity without compromising engineering integrity or flexibility" [3]. This also means that Roo is not restricted to Web application development and hence, is no WAF in the true sense of it. However, as section 4.3.1 will show, it is directly influenced by Web application development and people focus on using the tool in such an environment, either to develop *Spring MVC* or *GWT*[2] applications, the two inbuilt Web technologies [2].

Spring Roo want to address the configuration burden Java developers are typically faced with in enterprise development [27]. The tool configures and manages the infrastructure of the project, manages the build process and also tries to help in solving particular business problems [27]. Because of this focus on configuration, and the delegation of work to other frameworks and libraries, applications developed using Roo do not even require any Roo-specific libraries at runtime. For developers familiar with the Java Standard and Spring, learning Spring Roo comes down to learn a few new annotations and to get a feeling of how to use and communicate with the Roo shell.

In the course of this work, Roo was used to create Spring MVC Web applications. Spring MVC is part of the Spring Framework, whose fundamental capabilities were already discussed in section 2.4.8.

4.3.1 Motivation & History

From all the technologies discussed in this chapter, Spring Roo is the most recent one: Version 1.0.0.GA was released in December 2009. However, the motivation for Roo goes back to 2005 [3]:

Ben Alex, developer at SpringSource, noticed a mismatch between the WAFs of that time and the rich domain models he was used to implement. He was forced to create lots of boilerplate code for each domain model and used different tools for this. Therefore Alex created a code generator called ROO (Real Object Oriented) to ease this work and be more productive.

Though ROO got positive feedback at some Spring-related conferences in 2006 and 2007 and, Alex had to concentrate on other projects in the Spring portfolio. The feedback of a few companies that used the tool nonetheless was used in a complete new tool Alex publicly presented in April 2009. The name was changed to "Roo" to stress the term was no longer an acronym.

As already stated above, Roo's goal is to "fundamentally and sustainably improve Java developer productivity without compromising engineering integrity or flexibility" [3]. Sustainability in this sense means that the architecture proposed by Roo is standard and easy to maintain, and that productivity is provided throughout the whole project lifecycle. In doing so, Roo tries not to impose any trade-offs on the project. At runtime, no Roo-specific libraries are required. During development, it makes no specifications on the editors or tools the developer wants to use. Even removing Roo support from the project is easy by comparison [3]. Furthermore, the technologies it is build on are common Java technologies. However, the developer can always be sure that only libraries

required by the current project state will be part of the deployed application.

Concerning Roo's architecture, one can differentiate between the *Roo core*, the *base add-ons* and *third party add-ons* [3]. In contrast to the latter, base add-ons already come shipped with the Roo download.

At the time of writing, the latest stable release of Roo was 1.1.1. It is published under the Apache License 2.0.

4.3.2 The Roo Shell

The only thing that makes up Roo is a command-line tool called the Roo shell. Even the annotations that come with Roo are just another way to communicate with the shell. By leveraging the shell's capabilities, Roo can be used to create Web applications in a short time. This is accomplished by scaffolding and code generation approaches for specific frameworks. That is, Roo can also be regarded as an abstraction layer over usual WAFs.

Currently, most developers use the tool to develop Spring MVC applications [3]. The programming model of Spring MVC is very similar to the one used in Grails (see section 4.2.3). This is because Grails uses Spring MVC as its underlying WAF [65]. Therefore, an example will not be provided here. However, a typically controller method signature is depicted in section 5.3.3 on page 75. Besides Spring MVC, Roo also offers inbuilt support for GWT.[2] There also exists an add-on to create JSF 2.0 applications.[27]

Though being implemented as a command line tool, the Roo shell provides an unusual level of usability. It is designed to be usable without any study of the documentation, following a "learning by doing" approach. For example, this is the first output of Spring Roo before creating a new project:

Listing 4.9: Exemplary output of the Roo Shell

```
    ____  ____  ____
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.1.1.RELEASE [rev 156ccd6]

Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
roo> hint
Welcome to Roo! We hope you enjoy your stay!

Before you can use many features of Roo, you need to start a new project.

To do this, type 'project' (without the quotes) and then hit TAB.

Enter a --topLevelPackage like 'com.mycompany.projectname' (no quotes).
When you've finished completing your --topLevelPackage, press ENTER.
Your new project will then be created in the current working directory.

Note that Roo frequently allows the use of TAB, so press TAB regularly.
Once your project is created, type 'hint' and ENTER for the next suggestion.
You're also welcome to visit http://forum.springframework.org for Roo help.
roo>
```

Listing 4.9 already illustrates two important shell features: *Contextual Awareness* and *Hints*. In the present situation no project exists yet, so Roo can anticipate the first

[27] Project Site: `http://kenai.com/projects/Roofaces`

thing to do is to create one. Accordingly, when the user types `hint`, the information displayed is all about project creation. Though contextual awareness may appear very simple in this specific case, it is of great help throughout the whole project cycle and - in combination with the verbosity of hints - a main reason for this command line tool's usability. Additionally, different colors in the output guarantee a better readability. The input is further simplified by Tab-completion and Inline-help to find out possible options.

Possible commands encompass ways to backup the project, define persistent entities, add finder-methods to them, create unit, integration or Selenium tests, scaffold controllers, implement a JMS infrastructure and much more. Another very useful feature is the possibility to record a sequence of commands to replay them in different projects or on different machines. This approach is also used in all sample applications that ship with the Roo: Before executing their scripts using the Shell, they only consist of simple text files.

Furthermore, it is important to note that no unnecessary libraries are part of the application at any time. For example, a project does not become a Web application before a Spring MVC controller is added. This kind of dependency management support leads to smaller archive files and faster application startup [3].

The Roo shell is designed to operate during application development, that is, also when it is not used for typing in commands. By monitoring the project's file system, Roo acts as a central coordinator: Specific modifications of source files cause it to adapt or delete `*.aj`, configuration or markup files. This way, Roo is fully *round-trip aware* and can carry out many steps that would else be redundant and repetitive to the developer [3]. Roundtrip-awareness is provided irrespective of the IDE or editor the programmer uses in the current development phase.

All annotations that ship with Roo only exist to be observed by the Roo shell whenever they appear in the source code. Therefore, they are all source-level annotations, meaning they do not become part of the compiled `*.class` files and are only retained in the source code itself (this is also the reason why no Roo-specific libraries are required during runtime). Whenever a new instance of an annotation appears or an existing one is modified or deleted, this results in different actions carried out by the Roo shell. For example, if the developer adds the `@RooToString` annotation to a class, Roo will automatically generate a method following Java's `toString()` approach, taking all fields defined for the respective class into account. There are multiple annotations for different purposes, for example to adjust scaffolding, persistent entities or Java Beans. The annotations can be configured in a fine grained manner using annotation arguments.

Implicit communication with the shell is probably the one that is taking place most of the time. This follows Roo's philosophy of providing productivity throughout the whole project lifetime [3]. Another important guideline in this philosophy is that Roo "works the way a reasonable person would expect" [3]. This means for example, even though Roo as a code generator is running in the background, a developer does not need to fear Roo might make changes to files he is working with (unless the developer tells Roo to

do so). Section 4.3.3 will explain how this is accomplished in Roo. Furthermore, Roo is designed to forgive mistakes. If a developer is working on the project and forgets to start the shell, Roo will catch up all changes the next time it starts. Besides, Roo always runs in a transactional context that allows automatic rollbacks in case the developer revokes certain actions [3].

4.3.3 Special Approaches to Productivity

Many of the features the Roo shell provides make excessive use of AspectJ's ITD approach. This concept is therefore explained in detail in the following. To foster the productivity gained by Spring Roo, one might also use the SpringSource Toolsuite IDE. This is a special IDE offered by SpringSource which provides special support for many Spring-related technologies, including Roo.

AspectJ Inter Declaration Types

One of Roo's core principles is to be non-intrusive [3]. However, sticking to this principle becomes difficult in plain Java when there is code to generate. For example, in Roo a developer can annotate a class as `@RooJavaBean` to let the shell generate Getter and Setter methods for each field defined in the class. Furthermore, for any new field added in the following, the methods also need to be generated instantly. In common Java, the only solution to code generation in this case is to add the methods directly into the class containing the respective fields. Following this approach, Roo would need to modify class definitions the developer is currently working on. Besides requiring the developer to reload each file after Roo has carried out its changes (which, in the example above, would mean after each new field defined), this mechanism can easily interfere with unsaved code changes.

To avoid this misconception, and hide generated code, Roo makes heavy use of AspectJ's Inter-Type Declarations. *AspectJ*[28] is the de-facto standard in the Java world to implement Aspect Oriented Programming. What is most important about AspectJ here is that it is not only a library but an extension to Java. This way, the AspectJ language can provide additional functionalities to Java, but of course requires special bytecode weavers to result in valid Java programs.

The use of AspectJ is also the reason why not all IDE's are able to compile Spring Roo projects. This also causes features like content-assist to fail regarding Roo generated methods and fields. Calls to these elements will be regarded as errors. That means, to work with Roo inside an IDE, a compiler capable of *JDT weaving* must be available.

One of the possibilities AspectJ provides are Inter-Type Declarations (ITD). ITDs are files that allow the definition of class members and structures (fields, methods, constructors, even interfaces or superclasses) that are owned by other types [3]. For the example above, this means that the Getter and Setter methods can be generated inside an

[28] Project Site: `http://www.eclipse.org/aspectj`

ITD and outside of the original source file. Since both files get merged at bytecode level, types defined with use of Inter-Type Declaration (ITD)s behave just like normal types.

Roo also uses ITDs to provide a mechanism similar to Grails' dynamic finders. The shell makes specific suggestions what kind of finders could currently be generated. The ITD of such a finder looks like this:

Listing 4.10: Exemplary ITD used to generate a *finder*

```
1 privileged aspect ProductFinder {

3     public static Query Product.findProductsByNameEquals(String
          name) {
4         if (name == null || name.length() == 0) throw new
              IllegalArgumentException("Name argument is required");
5         EntityManager em = Product.entityManager();
6         Query q = em.createQuery("SELECT Product FROM Product AS
              product WHERE product.name = :name");
7         q.setParameter("name", name);
8         return q;
9     }

11 }
```

Listing 4.10 represents a finder for a `Product` entity. The bytecode weaver will add the method at compile time, therefore the developer is able to use this generated query method just as if it was defined right inside `Product`. Though working with a statically typed language, Java developers this way get a correspondent to dynamic finders known from frameworks like Grails or Ruby on Rails.

SpringSource Toolsuite

The SpringSource Toolsuite[25] (STS) is freely obtainable since May 2009. Before, it was only available to direct customers of SpringSource.

STS is in fact an Eclipse[24] distribution providing preconfigured and attuned plug-ins to support software development with Spring technologies. For example, there exist graphical Spring configuration editors or online XML validations. The Dashboard gives access to Spring-related blogs and news and provides tutorials to the different Spring technologies. A special feature is the architecture-support plug-in that analyses the project state and gives advices to the developer. Since the Spring framework has become a very large and fast growing project, this tool is especially useful to learn new features and get rid of deprecated Spring usage mechanisms. Exploiting this plug-in, the project will more likely follow best practices in Spring development.

However, since both Roo and Grails are both SpringSource projects, there also exists support for them: There are wizards to automatically generate Roo or Grails projects. Both the Grails console and the Roo shell can be integrated into the IDE, which is es-

pecially useful for Roo projects for they otherwise require a running shell in the background. Special views give a better overview on the current project state, e.g., about defined controllers or controller mappings. Concerning Roo's ITD concept, STS integrates the required JDT bytecode weaver and provides refactoring commands to "push in" code from ITD files into their corresponding Java object definitions.

Another IDE that integrates AspectJ support, the key enabling technology for Roo, is IntelliJ IDEA.[23] At the time of writing, Netbeans[22] was not able to deal with Spring Roo projects, for AspectJ support was missing.

Chapter 5

Productivity Criteria

According to their official documentations, all the frameworks analyzed in the course of this work are "rapid" (JBoss Seam), "rapid" and "agile" (Grails) or simply "productive" (Spring Roo). By examining the state-of-the-art for the productivity in web application development, this chapter should also make these vague terms clearer. Besides, the resulting catalog can serve as a guideline in WAF evaluations, e.g., regarding new web development projects.

To this, the first part of the catalog is based upon the Model-View-Controller pattern (see section 2.3). That is, section 5.1 deals with WAF support regarding the Model, 5.2 with the View and 5.3 with the Controller component. In the second part, productivity is examined regarding different software engineering disciplines or tasks: Section 5.4 covers development support in general, independent of specific artifacts, section 5.5 is about Builds and Integration and section 5.6 deals with Testing.

5.1 Model

Because persisting data is central to nearly all enterprise applications [4], persistence is an area addressed exceedingly in WAFs' approaches to productivity. Concerning the MVC pattern, the data that makes up an application is part of the Model.

5.1.1 Data Source Connection Configuration

The first step when communicating with a database is to establish a connection. In Java, JDBC is used to access databases. Therefore, to use a specific database, a JDBC driver must exist. This driver is usually provided by the database vendor. An application making Java Database Connectivity API (JDBC) calls therefore requires the respective driver (usually a packaged JAR) in its class path and needs to be told a class inside this driver that implements the `javax.sql.DataSource` interface. Next, the actual connection

properties need to be defined. These depend on the database and its configuration, but will usually contain information about the database URL and the credentials. Enterprise applications have special requirements where and how this information must be stored. Modern WAFs are built on ORM tools which typically introduce additional configuration aspects. Productivity in this area can be fostered in different extents:

In a first step, for WAFs usually also provide build scripts for compilation and packaging, a WAF should place the required configuration files to their intended place in the file system where the user then needs to fill in the missing pieces of information. Next, the user may be prompted to specify a path to the required JDBC driver. The same way, information about the connection may be provided to the framework.

Spring Roo simplifies this process by directly asking for the database and persistence provider. It then automatically obtains the JDBC driver and the default settings for the respective database and the persistence provider. Additional information that cannot be determined by defaults, like the database schema or user credentials, may be provided in the same step. This way, the database and the ORM framework often can be configured within a single command passed to the shell. However, this approach obviously can only work for databases the WAF is familiar with. When providing a feature like this, it is important it can be reused anytime during development to redefine the database connection settings.

Grails follows a different approach. The WAF automatically integrates and configures an HSQL in-memory database for the development and test environments (for environments, see section 5.5.2). Additional configuration is not required. This is especially useful in very early project stages or when evaluating the framework for the first time.

However, database configuration does not apply very often during development. It usually has to be defined only once or once per environment. Therefore, big productivity gains should not be expected here. The mechanisms defined in the next section have more abilities in this regard.

5.1.2 Reverse Engineering the Database

As already mentioned, Object-Relational Mapping (ORM) is an often-used technique in WAFs to provide access to relational databases (see section 2.4.6). Here, the design of the database schema is represented by persistent entity classes. However, software development projects nowadays often have to deal with legacy databases [71]. In this case, the database schema is already given and, to a large extent, defines the form of the entities. Reverse engineering describes the process of generating the entities from the schema information stored in the database.

Reverse engineering a legacy database works by pointing the framework to the respective database schema. Often, this is the same data source that was already defined for the Web application itself, so this step might be skipped. Every JDBC driver implements the `java.sql.DatabaseMetadata` interface, which is used by reverse engineering

tools to access information concerning the database's capabilities, its behavior and about the particular schema. This process is called *introspection*. The information gained here can then be used to finally create the persistent entities.

The advantage of using the WAF's abilities to reverse engineering (instead of a third party tool tailored for this requirement) is that the generated classes already stick to the conventions claimed by the WAF. Furthermore, the WAF this way is able to generate additional code for working with the entities, like Data Access Objects.

Spring Roo even provides an *incremental reverse engineering* strategy. That is, the tool repeatedly re-introspects the database and compares the result with the metadata currently expected by the application. If certain tables or columns have been deleted or renamed in the meantime or didn't exist before, the model is updated automatically, including possible references to the respective types. However, using an annotation-based approach, the developer can configure whether entities are allowed to be deleted this way (on a per-entity basis).

Reverse engineering might face the developer with multiple problems. Databases often use naming conventions inadequate for objects and their fields. While this problem might be by-passed by WAFs familiar with the respective database, there is no way to provide a solution for poor names introduced by other developers [4]. Furthermore, some information required can't always be gathered from JDBC metadata alone; the other way round, the developer might want to ignore parts of the metadata information [7]. To address all these issues, the WAF should provide ways to control the reverse engineering process. Otherwise, falling back to a professional 3rd party tool for reverse engineering might be the better option, for manual changes require a deep insight into the used technologies and, depending on the extent of the domain model, waste a lot of time.

Currently, Spring Roo's approach regarding reverse engineering control is restricted to the inclusion or exclusion of tables which names match certain patterns. JBoss Seam, in contrast, uses Hibernate's reverse engineering tool[9] in the background, which allows defining custom reverse engineering strategies.

5.1.3 Top-down Development Support

Naturally, a WAF also allows top-down development, that is, defining entity classes first which then get mapped to the database in a second step. All framework examined in this work offer ways to create stubs for new entities, accessible through the WAF's generator. How important such funtionality really is depends on the programming model and the way entities get defined in the respective framework. For example, JBoss Seam sticks very close to the JPA specification (see section 2.4.6), whereas Spring Roo prefers to split entites into multiple ITD files. Accordingly, for a developer familiar with the Java EE stack, defining an entity in Seam will be straightforward, whereas the programming model in Roo is specific to this particular tool. A command to automatically generate entity stubs therefore seems more important in the latter case.

This argument may also be the reason why Spring Roo provides a very fine-grained approach concerning entity generation through its shell. Here, it becomes even possible to specify single fields, their designated types and names, or relationships with other entities. This approach also offers interesting options regarding the validation of domain classes, to be discussed in the following section.

Next, when creating an entity using the project generator, WAFs may automatically add `id` and `version` fields to the generated classes. This is also an architectural support for adding these fields to every single entity can be concerned a best practice. Version fields are an important concept regarding database access and optmistic locking strategies.

Frameworks like Grails, which are based on dynamic languages, may automatically turn entity fields into properties, that is, making them accessible by means of Getter and Setter methods. Through its ITD concept, Spring Roo offers a simliar concept, turning classes into JavaBeans transparently. Though the time saved by such approaches cannot be regarded critical to a web project, such features still reveal useful since they automate steps that are often required by the underlying technologies or the WAF itself, but are easy to oversee due to the inherent inversion of control mechanism.

A completely different way of influencing the generation of persistent entities is to modify the underlying templates used in the background. Therefore, the respective templates should be provided by the WAF. However, the opportunities given this way are very limited and mainly address the visual appearance of classes. Template adaption will be discussed in depth as a part of the View (see section 5.2.1), for it is a much more important object there.

5.1.4 Validation of Entities

Validation is an important topic in web applications, as it is in any application accepting user input [47]. Sadly, it is also a topic often ignored by programmers [15].

Every WAF will somehow provide a way to allow validation of entities. At some point in request-processing, the control is always handed over to the application. This is where the developer can integrate validation by hand, at the very least. However, modern approaches try to separate validation concerns from functional code, in a declarative and therefore less error-prone and faster way. A WAF should support a similar approach and do the entire necessary configuration beforehand.

In a next step, the reverse engineering process should already concern validation and add the constraints derivable from the JDBC-Metadata automatically. If the framework also provides a way to generate single entity fields, then this is the point where validation support can be applied in a very fine-grained manner, on a per-field basis. For example, when using the Roo shell to add instance fields to an entity, validation annotations can be added in the same step. The contextual awareness of the shell makes sure that only relevant annotations are proposed. Since the Bean Validation API (see section 2.4.4) lists

more than a dozen annotations and Hibernate Validator[29] as a possible provider of this standard adds even more, such an approach makes sense: Otherwise, the developer might forget to add certain restrictions. The only way out would be a study of the corresponding documentation(s).

If a WAF provides its own validation mechanism instead of following a specification, there are a few rules this solution should follow to be productive: First of all, simple validations (like `min` or `max` values for numbers or the matching of given regular expressions against Strings) should be allowed to set in a declarative manner; otherwise, if the mechanism comes down to manual checks using assertions or `if`-conditions, there is only little gain for the developer. Second, the mechanism should still be flexible enough to allow the definition of custom validation constraints that can be bounded to the entity, that is, separated from application logic itself. All these points are satisfied by the validation approach used in Grails. Section 5.3.4 will list further requirements with respect to the invocation of validations.

CRUD pages, discussed in section 5.2.1, may foster validation support by automatically adding validation checks to the view, potentially even by using an AJAX-based approach, or use them to determine adequate HTML elements in web forms.

5.1.5 Entity Lifecycle Management

Promoted by Ruby on Rails [4], the Active Record design pattern has become a very popular design pattern in WAFs. Grails and Spring Roo[30] both offer an implementation. An Active Record encapsulates domain object concerns (as simple persistent entities in the JPA do) and data access logic in a single artifact. That is, the domain object is able to save and retrieve itself from the database [4]. Active record is concerned a faster way to handle persistence. For example, to store an instance to the database, one simply has to call `instance.save()`. There is no need to obtain a Persistence Context directly or to use a DAO. To implement the Active Record pattern, Grails uses Groovy's Metaclass concept and Spring Roo, for bringing an adequate approach to the static Java world, provides the @RooEntity annotation that triggers the shell to generate an additional ITD class for these purposes. This way, additional methods to find, persist or flush, merge or remove a certain entity become available.

However, the Active Record pattern also has some significant drawbacks. Most important is the missing separation of concerns, where domain models are hard linked to the persistence framework. Reusing these objects in XML serialization or for JMS is not possible without any further ado. Other problems may arise from the fact that Active Records hide the way persistence in handled in the background. Many use cases require to change multiple objects in a single unit of work and in a number of steps, guiding the user over mutliple screens until this conversation can finally be commited (which is

[29] Project site: `http://www.hibernate.org/subprojects/validator`

[30] At the time of writing, there was the open issue ROO-301, categorized major, asking for support of additional ways to access the database for not being forced to implement Active Record.

about to be discussed in section 5.3.5). This requires adequate persistence context propagation and management. Using the data access logic of an Active Record may then lead to unnecessary database communication, or worse, stale data. This is the reason why JBoss Seam offers the reusable `Home` component template for managing a single entity. Such a component can always be scoped and tested appropriately; however, for simple scenarios, using it is not as easy and fast as Active Records are known to be.

5.1.6 Database Queries

Querying the database basically faces the developer with two distinct concerns: The query must be formulated (in such a way that the correct results are returned in a performant way) and the respective result sets must be managed somehow. Both tasks may be supported by a WAF.

As the last section should already have shown, frameworks based on dynamic languages can easily intercept method calls and create code dynamically, just like Grails can using Groovy's `MetaClass` and `methodMissing` features. Concerning database access, it this way becomes possible to create queries dynamically at runtime, a feature called *Dynamic Finders* in Grails. Dynamic Finders search the database according to different criteria, including LIKE or BETWEEN searches. However, concerning Grails, one restriction occurs: Criteria might be joined using `AND` or `OR`, but querying is only allowed on two fields – and a single entity.

The provided finders usually include the following criteria, which should be self-explanatory:

- `LessThan`
- `LessThanEquals`
- `GreaterThan`
- `GreaterThanEquals`
- `Like`
- `Ilike`
- `NotEqual`
- `Between`
- `IsNotNull`
- `IsNull`
- `And` - for combining different criteria
- `Or` - for combining different criteria

Spring Roo shows that it is possible to provide a similar feature for non-dynamic languages too. Here, the developer sets the Shell's focus on a certain entity, may ask it to make assumptions on possible finders, and let it generate certain Finders in a special ITD. In contrast to Grails, Roo allows an unlimited combination of search criteria and fields. Generated finders can then be called like any other method defined for this entity.

That is, though still named *dynamic finders* by Roo, those methods become part of the static program code and are by no means dynamically generated at runtime. Therefore, the finders in Spring Roo will sometimes be referred to as *static finders* in the following, to distinguish them from the related concept used in Grails.

The Roo approach, though exploiting Aspect Oriented Programming features, even has an advantage over Dynamic Finders: Since the generated query code becomes available to the developer, one can take and extend it to address more complex queries. In contrast, the underlying query of a Dynamic Finder in Grails gets generated at runtime and is therefore not accessible. The concept is therefore only useful when the respective finder matches the required query exactly.

When executed, the finders generated by Roo and Grails always return the complete result set of the respective query. This behavior is straightforward to the developer, but at the same time, may put unnecessary pressure on the database. For example, a user searching for a particular data record may be interested in the latest results first. Querying the database for all data records, irrespective of their age, is inefficient then. Instead, when the age of the record is already considered at query time, it may be sufficient to return a much smaller result set in the beginning. A WAF should support such use cases, for a performant implementation can be a complex task and requires a deeper insight in how to handle the persistence context right. Seam, for example, provides the `Query` template component for contextual queries. Such a component is initialized with a query and offers different parameters to control the database access in a fine grained manner. Results can be paginated and therefore, no unnecessary rows need to be queried. As long as results aren't considered dirty by the component (which may happen when a different result page is requested or due to changing sort orders), the query is not executed again. This approach shows how result sets can be managed in a performant yet user-friendly and fast way.

5.2 View

According to the MVC pattern, the View is responsible for displaying information to the user, in particular the one stored in the model. Again, a Web Application Framework may provide some features here to ease the creation and management of views.

One important thing to note about the View is that its final design is usually made by a Web designer, that is, a person who is typically familiar with presentation standards (see section 2.1.2) but knows little about software development or programming as a whole. Therefore, a developer should always try to create views that keep close to the standards and contain as little programming constructs as possible.

5.2.1 View Generation

Concerning the presentation layer, the probably most time saving feature that can be provided by a WAF is the possibility to generate views automatically, a functionality also known as *scaffolding*. Scaffolding can be further divided into dynamic and static

scaffolding. WAFs may even offer round-trip awareness for the View, that is, modify a generated view automatically due to changes made by the developer in the underlying artifacts. Usually, the generation of view markup is based on some predefined templates that ship with the framework. By modifying these templates, scaffolds can be adjusted to the project accurately.

CRUD Pages

CRUD is an acronym for the four database operations *Create, Read, Update and Delete*. CRUD Pages are WWW-documents that provide a User Interface to this kind of operations. A WAF often provides ways to generate such CRUD pages automatically within a few seconds. The functionalities may encompasses ways to list all instances of a specific entity, split the results into multiple pages (a feature also known as *pagination*), view the details of a listed instance and its related entities (an operation also called *drill-down*), search for certain entities by specifying the contents of one or more contained fields (using `AND` or `OR` concatenations), sort the results by different categories, add new instances to the database by specifying fields and relationships to other entity instances, or to delete or edit specific entities. Table 5.1 lists common functionalities made available using scaffolding techniques.

Table 5.1: Common functionalities of generated Scaffolds

Function	Description
`Create`	Offers a web form to create a new entity and persist it to the database. For each property, a form field is provided. Solutions differ in the way they allow to create relations to other entities and in the way they take validation (see section 5.1.4) into account.
`List`	List all entities of a certain type. Screens may be cleaned up by providing pagination functionalities. Sorting the list by selecting a particular property may not always be possible.
`Read`	Show the details of a particular entity. Usually, this screen is accessed by drilling down from the `List` or `Search` screens. Solutions may vary in the way related entities are displayed.
`Search`	Provides a search mask to find entities of a specific type. Search criteria may be matched to certain fields and combined in an AND or OR manner. `Search` may not be provided by every WAF.
`Update`	Edit the values of a certain entity. The same restrictions as for `Create` may occur. Usually, this screen is accessed by drilling down from the `List` or `Search` screens.
`Delete`	Offers the deletion of the depicted entity. The functionality may also be accessible form the `List` or `Search` screens.

Differences in the scaffolding solutions typically address `Create` and `List` (and the related `Update` or `Search` screens, respectively). Concerning the creation of entities, the way field types and validation-restrictions are taken into account is crucial. For example, if an entity's `String` field accepts a large amount of characters, implementing a `textarea` instead of a simple input field may be appropriate. The same holds true for `Date` objects and JavaScript calendars that support their definition. Required inputs may be pointed out automatically, on basis of the information stored in the underlying entity. Fields can be validated instantly using an AJAX-driven approach. Error-messages might be displayed at a single point of the page, either above or underneath the respective form, or multiple error fields exist, each one next to the form element they are related to, in a more user-friendly fashion. Another question is how relations between entities can be defined, and for which types of relations (1:1, 1:n, n:m).

Respecting lists, two important features scaffolds should provide here are *pagination* and *sorting*. Pagination allows splitting up the overall result set into multiple pages. Besides providing a much better overview, this can significantly reduce the database load, for only chunks of data need to be accessed. Sorting eases the handling of the list: On the basis of the defined columns, records can be ordered in an ascending or descending manner.

CRUD functionality may prove useful for several reasons: First of all, listing or searching are common use cases in web applications. The generated views and controllers serve as a good starting point to implement additional requirements specific to the application under development: It is easier to delete certain undesired functionalities than starting to implement the use case from zero. If the view templates used to generate the screens have been adapted beforehand (more on this in section 5.2.1), even these modifications may not be required.

Second, CRUD generation proves very useful in situations where the resulting documents are not intended for public usage but for administrating the application's underlying database, e.g., to add or delete certain users by hand. Such *admin pages* are another use case common to many web applications and would probably take days of implementation without this feature.

A last advantage of CRUD generation is that developers new to the WAF can use the generated application to get familiar with its programming style and conventions. This way, the feature also serves as documentation for the best practices used in the corresponding WAF.

As shown, WAFs may differ widely in the way they scaffold applications. Therefore, when CRUD functionality is required or the WAF is only used to generate some admin pages, a comparison may pay out. From the frameworks examined in this master thesis, JBoss Seam provided the best solution, for it offered search functionalities, pagination and sorting, and was able to deal with all types of entity relations.

Dynamic Scaffolding

Dynamic scaffolding describes the WAF's facility to create views and controller actions on the fly, when the web application is deployed. That is, the generated views and controller actions aren't accessible to the programmer. For real-world applications, this is obviously no option. The functionality may still be sufficient for admin screens used to manage database contents though. However, the most important advantage of dynamic scaffolds is that they allow the developers to adapt the underlying database model over and over again, without being forced to carry out the same changes in the corresponding views. The generated views can then show whether the changes lead to the designated results or not. That is, dynamic scaffolding provides useful in the very early stages of a web application project.

It can be argued that a WAF that provides a mature round-tripping approach for the View (which is about to be described later in this section) can forgo dynamic scaffolding without influencing a developer's productivity. The other way round (i.e., replacing round-trip awareness by dynamic scaffolds), this is not true.

Without a modification of the underlying templates, dynamic scaffolding offers only a few options to influence the way views get generated. For example, it is possible to decide upon the ordering of fields that are displayed in forms. This ordering may adhere to the ordering found in the respective entity definition. In Grails, the ordering is derived from the fields' order in the `constraints` closure. Often, the underlying validation messages can also be adapted or extended for internationalization purposes. Validation is another aspect that may influence the dynamic scaffolds. The constraints for example can be used by the WAF to decide upon the type of fields that get displayed in a web form. This way, a `String` accepting more than 50 characters may result in a `<textarea>` element, whereas the default would be a simple `<input>` field.

The dynamic scaffolding approach of Grails works on a per-action basis: A controller administering a specific entity and configured for dynamic scaffolding will only generate views for those methods no controller actions are defined for. That is, the precedence of developer-defined actions is higher, which allows using dynamic scaffolding only for specific pages and stick to self-defined actions in the other cases.

Static Scaffolding

In contrast to the dynamic approach described above, static scaffolding is a feature that should be provided by any WAF build with productivity in mind. Here, the controller actions and views are only generated once and become a durable, that is, static part of the Web project.

Static scaffolding usually works by pointing the WAF to a specific entity. The WAF may additionally ask for the specific CRUD views the developer wants to generate, otherwise, the WAF will simply generate all of them. In the latter case, the additional overhead usually comes down to delete a couple of files by hand. Often, project generators also offer a command to generate all views for all available entities automatically.

The reason why static scaffolding is so important is that many use cases in Web development are similar to those offered by CRUD screens, at least to a certain degree. For example, a typical use case when providing a search function is that the results should be displayed in a list; therefore, generating a view and a controller capable to list all entities of a certain kind may be a good starting point (if `search` is not among the available CRUD functions). The "only" thing left would be to define a search field, read out the respective user input and modify the dataset returned to the view accordingly.

Even if the use case to implement does not match the general CRUD functionality, static scaffolding may still be useful for it automates many basic actions that a developer would have to do anyway. These tasks for example involve the creation of a new controller, associating it with a view, writing the view's markup by satisfying the possibly underlying template engine's needs, applying other WAF-specific configurations, and by all this, sticking to the conventions used in the overall project. Hence, static scaffolding can save a lot of time.

Round-tripping

The downside of static in contrast to dynamic scaffolding is that views usually remain untouched by the WAF after their generation. That is, the advantage of dynamic adaption to modifications in the underlying model gets lost. However, at some point of development of nearly every web application, a switch to static scaffolding is inevitable: The view must be customized manually to fit the actual use case. After that, each change of the entity also implies changing the corresponding view by hand.

The problem stated can be solved by using a WAF with round-trip awareness, that is, a WAF which is not only able to generate code or markup due to commands the developer has entered into the WAF's project generator, but also as a reaction to manual changes made in the corresponding files. One way to implement such behavior was already depicted for Spring Roo in section 4.3. However, Roo's general approach of reacting on the appearance/disappearance of certain annotations or instance methods by separating generated code from developer code in ITDs is not applicable for the View.

Instead, the following technique is used: All critical tags in a view are prepared with an additional `z` attribute. By default, `z` contains a hash-key calculated by the element's name, its attributes and attribute values (obviously, `z` as an attribute itself is not taken into account here). If the developer changes an element, this will result in a hash code mismatch. Since the only reason for such an event is that the user has modified the corresponding tag, `z` is automatically set to `"user-managed"` and Roo will not interfere with changes of this element. However, the user can replace `"z"` with a `"?"` anytime so that Roo will create a new hash code and take back the control. This approach is one reason why Roo uses the XML syntax of *JSP Documents* (see section 2.4.3) instead of common JSP.

Template Adaption

The View generated by a WAF always uses a default design and layout. For most web applications, this will not be enough: A unique layout is required by the application, adhering to the customer's corporate identity in page structure, colors, pictures and logos. Other reasons for an adaption may be needs imposed by the development team itself, for example, the integration of conventions or recurring information like the use of certain `<meta>` tags, legal details or website credits. Without being able to modify the predefined templates, a developer would need to adjust every freshly generated page to these requirements.

Templates consist of static markup code and instructions to the underlying template engine [24]. This template engine has to be distinguished from the one used in the deployed application for generating the actual view to the user. Often, different template engines will be used to fulfill these two tasks. For example, Seam uses Facelets as its default view technology to generate XHTML to the user, but the templates written to scaffold new web applications are based on *FreeMarker*.[31] The reason for this differentiation is that tools like FreeMarker are technology agnostic and can be used freely to create JSP or Facelets pages, Java or even Groovy code. In contrast, JSP is only able to generate Servlets.

The task of a template engine is to take a template and a data model and generate output based on this information. Adapting the scaffolding results therefore either works by adapting the template or the underlying data model. The latter is usually no option as it requires to go into the details of the template engine API and, more important, change the source code of the WAF generator itself. However, all required information is typically related to the database model and should always be among the information passed to the template engine. Hence, template modification is the primary way to influence scaffolds.

For the developer, it is important how the WAF makes the templates available. Here, two different kinds of requirements need to be distinguished: Some requirements may be valid for each new web application to develop, while others are application specific. Introducing application-specific requirements into templates that are used for every web application is no good idea, since the next application to generate will also contain these changes. The other way round, a recurring implementation of general requirements into every new application also consumes much time. Therefore, both global templates and ways to derive application-specific templates from them should exist. The application-specific templates then can be modified without interfering with other applications.

This approach is followed by Grails. The `install-templates` command makes templates available on a per-project basis. This is the place where application-specific changes can be made, on the basis of templates that already adhere to general requirements valid for all applications.

[31] Project Site: `http://freemarker.sourceforge.net`

5.2.2 View Composition

View composition is another feature that reduces the overhead of creating the presentation tier. Very often, web pages are made up of the same elements like headers showing the logo of an application, sidebars displaying the menu or footers containing copyrights or links to website credits. One can observe that, while surfing through the pages of a particular website, only parts of these elements do actually change, while others stay the same from screen to screen. That is, the screens share the same structure where the independent parts are always placed at the same position. This kind of consistency between different views is what a user is typically used to and is probably even a factor influencing the trustfulness of a web application. View composition strategies are based on this observation and try to reduce the markup code required to set up a new view. This way, they bring the *Don't repeat yourself (DRY)* concept, an important principle in modern programming, to the presentation layer. Basically, there are two different approaches dealing with View composition.

The *Composite View pattern* works by composing multiple parts, all defined in different files, to define the complete web page. One representative for a templating framework build on the Composite View pattern is *Apache Tiles*[32], for example used by Spring Roo. Here, a *template* defines the overall layout of the page, while some *attributes* build the inner gaps to be filled by the application. An attribute may be another template, a simple String or a *definition*. A definition "is a composition to be rendered to the end user" [35], combining a template with filled attributes in some configuration file. This approach allows for a reuse of markup in multiple views.

Another way is to use the *Decorator pattern*. In this approach, the developer renders a page back to the user as always, but the page gets intercepted by a decorator. This decorator is aware of the way the respective page needs to look in the end, i.e., it knows which template to use in the given situation. This template defines those parts that can be taken from the page and have to be merged with the other contents of the template. A popular layout engine following the Decorator pattern is *SiteMesh*[33], for example used in Grails.

The two approaches have different impacts on the web application. The Composite View pattern has a slightly better performance since the pages do not have to be parsed as by the Decorator Pattern. On the other hand, each page must be explicitly defined and composed here, whereas a single decorator could even be reused by the complete application [35]. In the end, the differences between both patterns will not be critical to the project, and using the one the WAF offers by default is probably the best advice. However, some WAFs do not offer a templating engine at all and delegate the work to the underlying view technology. View technologies like JSP indeed offer ways to include certain files, but cannot keep up with template engines designed for such use cases. Using only the view technology's mechanisms can be concerned error-prone, but

[32] Project Site: `http://tiles.apache.org`
[33] Project Site: `http://www.opensymphony.com/sitemesh`

at least results in a lot of overhead for the programmer.

5.2.3 Internationalization

Web applications are typically accessed using the Internet, addressing people from all over the world. Depending on the geographic location and the social background of the user, the application should be displayed using the user's language and the common esthetic sensation in the respective region. Therefore, internationalization (i18n) is required.

For the developer, this primary means not to use hard-coded text messages in markup files or program code, but instead use message keys which are then retrieved from special resource bundles that store the localized message contents. A WAF should automatically detect the user's Locale, e.g., using the `Accept-Language` header sent by the user's browser. For the Locale set this way is not adequate in all cases, the application should also provide a way to select a language manually. The WAF could support such scenarios by providing a special view component for language selection. Moreover, a default locale should be configured automatically. The WAF should put the corresponding message bundle to its designated place in the application's directory structure. This helps the developer to define additional languages for the application.

For internationalization is such an important topic, many libraries and frameworks used by a WAF come with their own solutions in this field. This typically results in multiple resource bundles and makes internationalization hard to manage. A WAF might therefore provide a way to centrally access all these different message bundles. For example, JBoss Seam offers a special `messages` component to address this issue. This solution can even be extended with additional bundles.

Another problem is how to test Locales. Switching the browser's language might not be required if the WAF provides support here: Grails for instance, allows appending a special `lang` parameter to the URL. For the given locale identifier is stored in a cookie, all following requests will be displayed in the defined Locale automatically.

It should be noted that the actual formatting (e.g., date and time patterns, currency units) will usually be delegated to Java's internal Locale implementation and is hence no WAF-related task.

5.2.4 Tag Libraries

View composition provides a way to reuse page structures and page parts. Tag libraries aim at the same direction, though on a more fine-grained level, supporting the reuse of certain markup constructs implemented in a collection of tags. Why and how to use tags has already been defined in section 2.4.3. Just like a rich set of libraries is a quality characteristic for a programming language, the same holds true for view technologies and the tag libraries available here.

To a great extent, the amount of available libraries depends on the popularity of the

respective view technology. Favoring a WAF keeping close to the standards in this regard may therefore be a good decision. The developer can then choose from a rich set of alternatives, ranging from simple open-source libraries over rich HTML input components to commercial products. In the worst case, for a WAF that defines its own presentation layer, chances are that only a single library exists. If the required functionality is not available or is simply not solved in a contenting way, the only solution would be to define an own solution.

Besides built-in or third-party libraries, most WAFs also ship with additional tags specific for the use of the respective framework. For example, Seam ships with over 30 additional JSF controls. Most often, such tags are simple modifications of the standard tags enriched by some information to fit the WAF's needs. Sometimes though, they can really simplify the implementation of the pages, especially if the particular use case is among the WAF's "sweet spots" (for this term, refer to section 2.5.4). Therefore, additionally provided tag libraries may be worth a look when comparing different WAFs.

5.3 Controller

The Controller is the last component in the MVC pattern to talk about. It handles the incoming user input and, hereupon, calls the model's functional interfaces. The result of these calls may also influence the view to be rendered.

Controllers encapsulate big parts of the application's business logic. For business logic being very application-specific, the possible support a WAF can provide here is limited. Nevertheless, WAFs provide many features that can reduce the required development efforts.

5.3.1 Scaffolding

When the web application or parts of it are scaffolded by the WAF, this does also involve the creation of controller code. While generated entities or views often adhere to standards well-known to the developer (e.g., JPA resp. JSP or Facelets), controllers are highly specific to the WAF in use. This is where another advantage of Scaffolding as a concept becomes obvious: A developer new to a WAF can use the generated code for learning purposes to get a feeling for how to use the respective framework. The knowledge gained this way can be helpful to manually define new controllers and stick to the best practices propagated by the WAF developers themselves.

As stated in section 5.2.1, WAFs differ in the extent they offer scaffolding to the developer. Besides providing scaffolding techniques as a start, it is most important that these techniques remain usable over the complete development phase. This statement is not only valid for controllers, but also for generation of entities and views. Frameworks may decide to offer scaffolding only when the application is set up and the model is generated from the database the first time.

Offering commands that distinguish between controller-plus-view generation or controller-without-view generation may be useful; however, if this feature is missing, the resulting overhead comes down to deleting some files manually. The same holds true for the question whether the WAF always generates a complete CRUD-able controller or allows for a more detailed definition. In this case, depending on how much functionality is actually needed, the overhead of adapting the project afterwards may be bigger though: For example, if the only possible action should be to list all entities, this may require to delete controller actions and views for creating, updating and deleting entities of that kind. Depending on the framework, even modifications to the model may be necessary to keep the code clean of unneeded functionality.

Section 5.1.6 already discussed the possibility to automatically generate finders to search for entities according to different criteria. Frameworks that offer such functionality may also offer ways to generate controller actions that call such finders and views which in turn call these controller actions. This way, providing search functionality to the user becomes a matter of seconds. For searching is an often-required use case, this feature may be of major importance for many projects. This scenario also demonstrates a disadvantage of dynamic finders offered by WAFs based on dynamic languages: For the required search functionality is not known at compile time (and provided by language features such as Groovy's `MissingMethod` approach), automatically generating views for them is not possible.

Usually, code generation approaches follow certain code conventions. For controllers, these conventions may for example concern URL and view mappings. However, since those conventions may not fit to the respective application or aren't appreciated by the development team for any reasons, the templates used to generate the code should be accessible to the developer. However, a differentiation between application-specific and global templates doesn't seem as important for Controllers as it is regarding the View.

5.3.2 Handler Mapping

In request-based WAFs, every incoming request is received by a Front Controller (see the Front Controller pattern in [36]). One of the first things this Front Controller, typically a Servlet, does is trying to map the request to a controller (this time, a controller according to the MVC pattern). To this, one or more so-called handler mappings are consulted. Such handler mappings define algorithms on how to map HTTP requests to controllers. Hereunto, different information from the request may be used,e.g., the URL and the HTTP-method. Component-based frameworks like Seam involve similar procedures.

WAFs should not mandate a particular approach here (Spring MVC, as an example, provides different pre-configured mappings and also allows to define custom ones), but via scaffolding techniques, they will often do, for the generated controllers adhere to the default handler mapping requirements. As long as developers cannot specify the desired handler mapping beforehand, this actually means that the WAF is responsible to provide a convenient, easy-to-use and flexible convention. Frameworks may use

completely different approaches here: For example, Spring Roo uses Spring MVC's `DefaultAnnotationHandlerMapping` to map requests to controllers and controller methods. The annotation-driven design allows a very fine-grained and RESTful[1] configuration where even request parameters and header fields can be taken into account. However, the generated Controller classes become highly dependent on this specific handler mapping. In contrast, Grails does not provide such a fine-grained mapping, but the controller code remains clean from any handler mapping definitions. This way, changes to the handler mapping can even be announced in a simple Groovy-script, without interfering with generated scaffolds.

5.3.3 Data Binding

After finding the appropriate controller method, the request is passed there. However, since the common `HttpServletRequest` API is cumbersome to use, most WAFs provide ways to ease parsing a request. These techniques are commonly referred to as *Data Binding*. Some of the techniques described in the following are either specific to request- or component-based WAFs.

Command Objects

When receiving data by means of Servlets, the mismatch between HTTP and Objects becomes obvious: Each single value needs to be picked from the request and probably also checked for existence before finally calling the Object's constructor using these values. Obviously, this is a time-consuming task and results in many lines of code. *Command Objects* or simply *Commands* provide relief in this regard: By binding the underlying web form in the view to a command, which has a code representation of a class, typically a persistent entity, the request parameters get automatically bound to an object representation by the WAF. This makes request processing easier and much more natural from an object-oriented point of view.

Command objects also address situations "when there isn't a one-to-one mapping between the form data and a domain object" [71]. A popular example for such a situation is a password verification field to protect the user from setting a misspelled password. Such an input has no representation in the entity representing the user. Instead, command objects can be used for marshalling the request data. Concerning Grails, converting command objects into their persistent entity representation after validation (which works the same way for them) is simple: Using Groovy's implicit constructor feature, which allows to set fields by passing a map of key/value pairs, and the `properties`-method, which is available for every class and returns the object's fields as a map, conversion comes down to a simple constructor call:

```
UserCommandObject uco = ...
def user = new User(uco.properties)
```

Keys in the map that do not match fields in the entity will simply be ignored. This way it becomes even possible to use the `params` object, a map-representation of the request parameters, to initialize objects.

Implicit Objects

The mentioned `params` object is one of the implicit objects Grails automatically injects into each controller instance. Besides `params`, which provides all request parameters in a map, there also exist implicit objects for the different storage contexts (see section 5.3.5) or logging purposes. For they get injected automatically, accessing them and using the features they provide is easy and fast. The transparent approach also results in a less technical, more functional code in the controller. In fact, controllers in Grails are another example of Groovy's power regarding Domain Specific Languages.

Dependency Injection

In pure Java, an equivalent concept to implicit objects is Dependency Injection (see section 2.4.1 for details). It is therefore also used in Spring MVC and Seam. In Spring MVC, the current session is one of the objects that may be injected this way. Besides the already discussed concept of bijection, Seam offers further annotations to inject various objects, e.g., single request parameters, loggers or information concerning data models. It should be noted that Dependency Injection is a less transparent concept than implicit objects, for a field has to be defined and, most of the times, some associated metadata. However, dependency injection allows for more manual control when to inject which types – a feature missing for implicit objects, resulting in a less performant solution.

Method Signature Analysis

To provide a convenient data binding approach, the WAF can analyze the controller method's signature and especially the defined parameters. Depending on the respective algorithm, this technique may also be used in the handler mapping. The following example of such an (annotated) method signature is taken from Spring MVC:

```
@RequestMapping(value = "/{id}", params = "form", method = GET)
public String updateForm(@PathVariable("id") Long id, Model m)
```

The `@RequestMapping` annotation's primary concern is to specify the method's metadata required for the handler mapping. In the example, a request will be passed to the method, if and only if (1) GET is used as the HTTP method, (2) a request parameter named "form" and (3) an additional path level (relative to the controller's primary mapping) are specified. Obviously, this controller method's task is to serve a web form to update some entity (probably the type of entity the controller is responsible for) with the `id` sent as a part of the URL.

When examining the method signature above, the only special characteristic compared to a common method signature in Java is its `@PathVariable` annotation. This

annotation is used to bind a method parameter to an URI template parameter specified in the request mapping; here, the id specified in the URL gets automatically bound and casted to the method parameter this way. Spring MVC also supports the `@RequestParam` and `@RequestHeader` annotations to do the same for request parameters resp. header fields.

Of course, the stated method signature is not specified in any interface or abstract class part of the Spring Framework. Spring MVC makes use of Java's reflection technology to be able to call such methods. The allowed types are listed in the Spring Framework's Javadoc for the `@RequestMapping` annotation.[34] This approach allows flexible signatures where only the actually required information needs to be specified. By additionally leveraging from autoboxing and annotation features, data binding becomes much easier than it is using plain Servlets.

5.3.4 Input Validation

Binding the request data to the Controller is not enough. The information received still needs to be validated. Input validation is an important topic often overlooked by programmers [15]. A WAF should try to ease this work and provide ways to keep the controller logic clean from validation concerns.

Spring Roo uses the Bean Validation API (see section 2.4.4) for validation purposes. The respective annotations can either be set using the Roo shell or by placing them manually in the code. Entities are automatically validated when `persist()` or `merge()` get called. However, for cases where the main processing is done beforehand, this is too late. And even though the object in question might never get stored to the database, other calculations can be based on the object which therefore still requires validation. To this, Spring MVC allows annotating controller method parameters with `@Valid`. At the same time, the developer can define a `BindingResult` parameter. Spring MVC then automatically invokes validation each time the respective method is called. The results are stored inside the BindingResult object, which the developer can programmatically check for possible validation errors.

Grails provides a similar validation support. Entities can be equipped with an additional `constraints` closure. As the name says, this closure allows defining constraints in a way similar to the declarative approach known from the Bean Validation API. Whenever the respective object is passed into a controller, the developer can call an entity's (automatically generated) `validate()` method to check for potential errors.

Section 5.1.4 already listed some requirements to be fulfilled by custom validation mechanisms introduced by WAFs. Concerning input validation, this list can be extended by the following points: The approach has to provide some API to inform the developer about the validation results in detail. Next, a decent error propagation mechanism should be provided, open for internationalization concerns. That means that error messages are inserted into the view automatically, using some kind of naming and configuration

[34] `http://static.springsource.org/spring/docs/3.0.x/javadoc-api`

conventions. If the WAF also offers automatic validation-triggering, as achieved by the `@Valid` annotation in Spring MVC method signatures, this is only an advantage as long as the mechanism is flexible enough to allow the developer to circumvent these automatic checks where required.

Concerning command objects defined in the last section, there is another problem related to validation and security of the application. Since form fields get automatically bound to commands by matching against the property name, intentionally modified HTTP requests may try to set properties in the command object that have not been specified in the web form and aren't intended for modification. WAFs that offer command objects therefore should also offer ways to circumvent such attacks. Possible solutions here are black- and whitelists: These lists either define those fields not to be modified (blacklists) or the ones that are allowed to be set (whitelists). This solution is also provided by Grails and its `bindData()` method for controllers.

One should note that not all validations can be defined on entity level because they are only specific to certain situations in the application flow. For example, even though a valid number is passed to the controller as an entity `id` for update concerns, this does not mean an entity with this id does actually exist in the database. That is, despite all kinds of automated validation, the developer should always remain critical concerning request data. Application logic cannot be totally clean of validation concerns.

5.3.5 Context Scope Management

After validation, the controller is finally ready to process the request. Obviously, the details of processing are application specific. At the very end, a view will be passed to the user. However, for this document to render, certain data may be required by the underlying view template. This is the point where *context* or *variable scopes* come into play.

Context scopes can store references to all kind of objects, for a specified period of time. This period depends on the respective scope used for storing. The Java Servlet technology (see section 2.4.2) defines three different context scopes: (1) The *servletContext* or *application scope* accessible by the whole application for its complete lifetime, (2) the *session scope* unique to a single HTTP session and (3) the *request scope*, which's lifetime is bound to a single request only.

For web applications, the limitation to these three scopes implies some significant shortcomings: Concerning one specific user, there are only two available scopes to store information in. Since web applications often involve use cases that require user assistance over multiple screens (i.e., multiple requests), the information required throughout the process needs to be stored in the session scope. However, storing such information in the session scope can easily become a problem. Compared to simple desktop applications, web applications have only little control over their usage. For example, the user can easily (and without any bad intentions) open a second browser window and start the same process again, e.g., for comparison reasons. Now, two parallel processes work

on the same context variables, interfering with each other. Prevention of such behavior is a very complex and time-consuming task; therefore, every WAF should provide additional scopes for solving such use cases. For example, Seam offers the *conversation scope* on top of a user session that distinguishes separate processes using conversation identifiers. Web flows are another approach to implement such scenarios. Leveraging these solutions, the developer is also disburdened of cleaning up the session manually.

Another problem is the very short life of request scoped variables. After sending the response, all information is dropped. This behavior conflicts with the *Redirect-After-Post pattern*, a best practice in web application development [49]: The pattern says that a view should never be displayed in response to a POST request. Instead, a redirect response code should be send, navigating the user to the final view using GET. This approach protects web applications against users reloading pages or clicking browser back/forward buttons. Without the pattern, such behavior could easily result in sending the same POST request again (which is problematic since POST is no idempotent operation, see section 2.1.1). However, between responding to the POST request and receiving the new GET request, the web application will drop all request-scoped context variables. That is, the view can no longer access the results of the previous request. Scoping such information to the session requires the developer to do a manual clean up afterwards, being not only a tedious but also difficult task. Modern WAFs therefore offer special scopes able to survive exactly one redirect, usually called *flash scope*. Regarding the amount of POST requests in an application, such scopes can save the developer a lot of time: A user posting data will at least expect to read a success or failure message. However, providing such scopes is not at all a matter of course in WAFs. For example, without using Spring Web Flows (which would be cumbersome for this kind of tasks), Spring MVC offers no such scope.[35]

Besides the availability of certain context scopes, it is also important how easy these scopes can be accessed. From the WAFs reviewed in the course of this work, Grails offered the simplest approach by making all scopes available through implicit objects. Furthermore, maps returned from controller actions get automatically passed to the view. In Spring MVC, controller methods may define `Map` or `org.springframework.ui.Model` parameters that can then be filled and later be used in the view (without returning them explicitly). Furthermore, a return value approach similar to the one used in Grails is provided. To use it, the controller method's return value has to be `@ModelAttribute` annotated though. If the developer wants to declare some session scoped data, the session needs to be injected into an instance field of the controller. This behavior is inconsistent with the approach used the for the request scope. Seam as a component-framework uses its Bijection mechanism (see section 4.1.2 on page 40) to read and write variables in the different scopes. Concerning `@In`, if no scope is explicitly defined, Seam will search through all scopes to find the respective variable, starting from the narrowest one.

[35] The integration of a flash scope in Spring MVC is scheduled for version 3.1 of the Spring Framework.

One should note that the concept of context scopes has nothing to do with an appropriate persistence context management. Even if an entity gets stored in a durable context (like flash, conversation or session scope) to be accessed in future requests, the entity might still get detached. This is definitely the case when JPA's persistence context is used in *transaction mode*, as it is the default when using a container-managed `EntityManager`. After the request has been processed, this will result in detaching all entities currently loaded. When accessing the detached instance in a later request, `merge()` has to be called to load the object into the new persistence context, required to propagate any of its changes to the database. However, `merge()` results in an additional database select and, even worse, it overwrites any database changes related to this object in between. This is the reason why calling `merge()` should be avoided where possible [4]. Instead, the persistence context could be used in *extended mode*, possibly together with an optimistic locking strategy. However, such use cases cannot be foreseen by a WAF. Having an easy access to the persistence context therefore is an important point for web applications. JBoss Seam, emphasizing its strength in stateful scenarios, is a WAF providing special support in this regard.

5.3.6 View-Mapping

The last remaining controller task is to define the view that needs to be rendered to the user. There are different techniques to fulfill this requirement. One idea is to apply a simple convention. The view-mapping in Grails for instance is based on the unqualified controller class name and the name of the closure called therein. Since each controller has its own folder of views, the framework will use this folder to look for a file that has the name of the respective closure. For example, if the `list` closure is called, the view to be rendered is `list.gsp` from inside the controller's folder.

Using conventions is the easiest way to define a view. However, it is not very flexible and therefore not always adequate. Another approach dealing with this issue is to return simple Strings that define the name of the view. The disadvantage is the now introduced dependency between the controller code and the view. However, since the usual relation between controller and view is 1:1 [36], this is less problematic. The only alternative would be to define the view to be rendered in a separate configuration file. The disadvantage of this solution is that the application flow in the code will often become harder to trace. In the end, this is also a matter of taste. Project teams should define conventions on how to define view mappings by default.

The last section already described the importance of redirects. Since redirects may occur quite often, depending on the amount of POST actions in the application, a WAF should provide easy mechanisms to initiate redirects. Grails offers a simple redirect method here that either accepts a String of the designated URL or controller/action references to call after the redirect. In contrast, the only places Seam allows to define redirects are the external page descriptors.

5.4 Development Support

The last sections covered the ways WAFs can address productivity gains concerning the different components defined by the MVC pattern. This chapter now examines those features that increase the developer productivity throughout the project, irrespective of the current artifact under development.

5.4.1 Project Generator

The project generator is a piece of software that comes bundled with a WAF. Many of the productivity features discussed in the course of this chapter are accessed by means of the project generator. However, it should be noted that the term *project generator* is no official term; in JBoss Seam it is called *seam-gen*, for Spring Roo it is the *Roo shell* and Grails does not even have a special name for it. Common to all solutions is that the development of new web applications always starts using this piece of software. For being a simple command line tool, the developer usually opens a console window to make a few inputs and create a first scaffold of the new web application. As the next section will show, some WAFs also offer UI-counterparts of their project generator in the form of IDE plug-ins.

Scaffolding and other features invoked from inside the generator have already been discussed before. This section talks about those features that characterize the project generator itself, defining the fundamental abilities that concrete features can build on.

The first question concerning project generators is how user input is handled. The three WAFs examined in the course of this work all provide different solutions here. One approach is to direct the developer through a defined set of configurations. Concerning the application's control flow, the user has only little or no influence at all. For many information is given to the tool in one such input cycle, the total amount of commands the generator accepts will usually be smaller (and therefore, easier to remember). However, this approach only seems feasible for setting up new projects. Functionalities required later in the project do not depend on so much information. Accordingly, this input approach should be coupled with another, more flexible one. JBoss Seam uses this approach in form of a questionnaire when creating a new web application.

A second solution is to design the generator in such a way that in each run, only one command is accepted. After processing the input, the generator stops execution. For issuing a new command, the generator software will need to start again. The advantage of this approach is its flexibility: Each feature-to-be can be encapsulated in a single command — it is all a matter of additional arguments for the respective command. However, it is difficult to hold state between two distinct commands. Some use cases, e.g., adding additional fields to generated entities, are therefore hard to implement. Furthermore, for the generator needs to start and stop over and over again, issuing a sequence of commands takes much longer than it would actually be required.

A last approach, proposed by Spring Roo, combines the advantages of both worlds.

Starting the generator means starting another shell, waiting for input commands. This way, for all commands are invoked within the control of the generator, handling state is much easier (though still complex when shutting down the shell and open it again). That allows for *contextual awareness*: Depending on the current artifact under focus, a special set of commands is available, while others are hidden by the generator. Besides increasing the clarity of the generator, able to deal with a huge amount of commands, and its general usability, this also allows questionnaire-like approaches known from *seam-gen*. Furthermore, the approach is flexible and allows an easy integration of new commands.

As it should be clear by now, many possible productivity gains in web application projects owe to the generator. However, features like static finders or view generation show that its work is by no means restricted to scaffolding a new application. Therefore, what is most important about every project generator is that it also allows for issuing commands after the web application has already been created. Seam shows a related shortcoming here: *seam-gen* can only be connected to one project at once – after creating a second web application using the tool, the first project can no longer be managed using the generator. The functionality provided by *seam-gen* (at least the one that is not part of the project's build file), is lost.

In contrast, the Roo shell can not only be used throughout the development phase, it is also able to do round-tripping between its internal state and the changes made by the developer outside the shell. The user can even communicate with the shell running in the background by means of special annotations. Adding or removing these annotations in the application code, or changing an annotation's attributes, results in code generations invoked by the shell. By adhering to certain conventions, in particular not changing generated ITD files, this roundtrip-approach does not interfere with the developer's work. For the shell does also observe files under control of the developer, it is possible to copy generated methods from an ITD file (which is permitted) and reuse it in the own code: When the shell recognizes the new method, it will automatically remove the generated one — no compiler errors claiming duplicate method definitions will occur.

Spring Roo also allows to record commands passed to the Roo shell in a special script file. Such scripts can then be called from inside the shell to repeat those steps. This way, repetitive tasks, especially during project setup phases, can be simplified or even exchanged between different computers.

Code-generation by the WAF's generator should always be preferred to own declarations. Besides saving much time in comparison, this is also because configuration and code then adhere to the best practices used for the respective WAF. Therefore, to foster its usage, the generator should also address the comfort of input. The Roo shell, for example, offers tab-completion, a `hint` command with context awareness and different text colors to increase the clarity of the user interface. Other simplifications address the usage of default values and the remembrance of values already used beforehand.

5.4.2 IDE Support

An Integrated Development Environment (IDE) is a software environment that covers a wide range of functionality to support the development of software. Though there are multiple IDEs available for Java, a recent survey in [75] has shown that more than 90 % of Java developers use one of three major IDEs: Eclipse,[24] IntelliJ IDEA,[23] and Netbeans.[22]

For WAFs, this is of big advantage: They need to make the web project and its structure known to the IDE; however, there do not exist standards in this regard. For each IDE to support, additional efforts are required. In their own interest, WAFs should make sure that at least the mentioned IDEs are able to deal with their web applications. Otherwise, features like compilation, content assist or on-click-deployment won't be available for developers sticking to the respective IDE.

In the course of this work, four major ways have been identified how IDE-support can be achieved.

The first option is to create IDE-specific configuration files using the project generator. There either exist separate commands to create them or the files are generated automatically when the new web project is set up. The last approach is followed by JBoss Seam: The framework creates IDE files for Eclipse, Netbeans and IntelliJ.

A second option is presented by Grails. This WAF has native support both in Netbeans and IntelliJ. That means, those IDEs are developed in a way to understand Grails projects automatically. Obviously, such an approach only seems possible for the most prominent WAFs.

Concerning Spring Roo, IDE support here is delegated to the underlying build tool Maven.[36] Maven provides multiple plug-ins to generate configuration files for various IDEs on the basis of the project information stored in the POM. This support has also been integrated into some IDEs.

A last option is to provide a project-generation plug-in for the respective IDE. Usually, this option comes as an additional alternative to one of the three approaches mentioned before. Regarding Seam, a UI for creating new projects has the further advantage that the input given otherwise in a questionnaire in the console can be immediately checked against flaws. However, it is important that common build files (see section 5.5.1) are still provided; otherwise, a lot of flexibility would be lost outside the IDE. An application should always remain manageable from inside and outside an IDE.

Whichever option is followed, in the end it should be possible to compile, package, deploy and, where available, hot deploy (see section 5.6.1) the application. As Spring Roo shows, even if the IDE is able to grasp the application's structure, this does not automatically mean it is able to compile it. Roo's ITD approach requires a special compiler capable of JDT weaving; otherwise, compilation will fail. This has already been discussed in section 4.3.3 on page 55.

[36] Project Site: `http://maven.apache.org`

Many WAFs also provide further tooling support for specific IDEs. Just like configuration files, writing such plug-ins is highly IDE specific. However, their implementation is obviously much more complex. A WAF therefore can provide IDE plug-ins, but better does not depend on them: IDEs, just like WAFs, tend to be updated quite often. Whether a new version will work with the old plug-in is not always guaranteed.

There is a variety of functionality that can be provided through IDE plug-ins: As already stated, a plug-in may provide a UI to scaffold new web applications (including reverse engineering the database). In contrast to simple build scripts, hot deployment, as far as implemented, can be initiated automatically every time a file is saved in the editor. For creating new entities, views or controllers, new entries might be added to the IDE's menu bars. Just like in the command line generator solutions, those new files are filled with stub code and are placed in the correct directories automatically. Special project views may be offered, providing a better outline of the complete application and the created artifacts. For these WAF-specific artifacts, plug-ins can offer an enhanced content assist. Concerning Grails, there even exists content assist regarding Groovy scripts (for Java code, this should be standard). Grails and Roo also offer counterparts of their command line tools in the IDE. This way, the developer no longer has to take care of the console. But it also allows the IDE to carry out workspace refreshs automatically, every time new code has been generated.

Another feature that can be addressed by plug-ins is error detection. For Java is a statically typed language, error detection in code is usually no problem. However, web applications do not only consist of plain code; usually, there are also markup or configuration files to be managed. Quite often, the content in those files refers to external elements, defined at another place in the application. One typical example are beans accessed in JSF views (see also the code listing in section 2.4.5): Spelling mistakes here usually aren't recognized until runtime. However, special IDE plug-ins can validate such inputs instantly.

Particular artifacts can also be supported using special visual editors. However, for writing such visual editors is a complex task, they are usually only available for the most common artifacts, like JPA entities of JSP views.

5.4.3 Dynamic Language Support

Dynamic languages are concerned a way to reduce the time to write application code.

> *"A dynamic language basically enables programs that can change their code and logical structures at runtime, adding variable types, module names, classes, and functions as they are running. These languages [...] generally check typing at runtime."* [60]

The code produced by dynamic languages is generally more compact [60]. One reason for this is that most dynamic languages are dynamically typed, reducing the overhead of type declarations. Moreover, dynamic languages often come with special operators

and other structural items that do not only reduce coding effort, but also increase a developer's flexibility in writing the application [60]. Section 4.2.2 exemplary lists some of the features provided by the dynamic language Groovy.

Besides Java, the Java Virtual Machine (JVM) also supports dynamic languages. For its task is to execute programs translated into bytecode, any language that compiles into bytecode can run in the JVM. As a consequence, many dynamic languages for the Java platform exist, Groovy only being one of them. Furthermore, efforts are made to ease their development and usage: JSR-223, part of Java SE 6, and JSR-292 (according to Oracle's current roadmap[37] scheduled for Java SE 7) simplify the access of Java code from scripting languages, provide a framework on how to host scripting engines inside Java applications, add special bytecode support and should generally let dynamic languages run fast on the JVM. [59]

According to [64], a typical Groovy application contains only 40-60 % LoC an equivalent Java program requires. Though sounding impressive, this hides some of the disadvantages inherent to dynamic languages and dynamic typing that can easily increase the effort for the developer. Above all, type-related errors aren't found until runtime, whereas static typing detects them at compile time. The overhead of fixing such errors is therefore much bigger in dynamic languages, and gets even worse for web applications where an application needs to be deployed to a server to run. Furthermore, dynamic constructs make applications harder to debug and often harder to understand. Though these issues can partly be addressed by writing tests, it is difficult to say how big the productivity gain by using dynamic languages really is.

Concerning WAFs, dynamic languages can be supported in different ways. In Grails, a dynamic language is the default to be used within the WAF. The whole programming model is aligned to the usage of Groovy, providing Domain Specific Languages (DSL) for controllers, persistent entities or service classes. In contrast, Spring MVC and JBoss Seam is based mainly on Java, though both also allow the usage of dynamic languages and the "syntactic sugar" typically provided by dynamic languages. For example, it is possible to define the persistent entities or controllers resp. components using Groovy. This way, the developer can benefit from certain language features. For example, mathematical calculations or the work on collections might be tasks appealing to delegate from Java to dynamic language code. However, what these solutions lack is the clarity of a DSL support. For instance, though getter and setter methods can be omitted when writing persistent entities using Groovy, the respective fields still need to be annotated - something unusual in Grails or Groovy as a whole.

Nevertheless, using dynamic or static languages is by no means an either/or decision. Instead, it makes sense to switch between both approaches, depending on the respective task. WAFs should at least provide a way to use dynamic languages.

[37] `http://openjdk.java.net/projects/jdk7/features/#f353`

5.5 Build & Integration Support

The directory structure and configuration of web applications differs widely from standard desktop applications. This section describes how WAFs provide support in this area.

5.5.1 Build Management

Supporting the build of a project should be a matter of course for every WAF. By using the WAF's generator for the first time on a new project, all required steps should be executed automatically.

The first thing to do is to create an adequate directory layout for the new project. There needs to exist a place where to store the source files of the project. These source files result in compiled classes which as well need to be put somewhere. The same holds true for resource and configuration files or required libraries. Test code should be separated from functional code by means of separate directories.

However, creating a ready-made directory layout is not enough. The WAF should also provide ways to execute build steps on the basis of these layouts. Typically, compiling and packaging the application will be supported, for both highly depend on the used directory layout and the places where configuration and resource files reside. Next, the deployment of the resulting archive could also be part of the WAFs build solution. Deployment makes an application available on a web or application server. Different kinds of deployment commands may exist in parallel, e.g., to support hot deployment (see section 5.6.1). Other targets may allow invoking the available unit, integration or functional tests (see section 5.6).

Usually, a WAF will use an existing tool for the management of builds. Tools like Apache Maven[36] already propose standard directory layouts for Java projects. When keeping to these standards, developers used to the respective tools will easier understand the project layout.

Tools like Maven also integrate solutions to define custom build steps. Grails (in its common usage) follows a slightly different approach here. For build commands are typically invoked using the project generator here, the developer can define groovy scripts that implement the respective build target, and either put them into the application or Grails installation's `script` directory. This makes the script part of the commands available through the generator, exclusively for the project or globally for all Grails applications.

Closely connected with the creation of a directory structure for the project is the generation of project files for specific IDEs. The generation of IDE files has already been discussed in section 5.4.2.

5.5.2 Environment Configuration

Throughout their lifetime, applications are used in different environments: In the beginning, during their creation, they are mainly accessed by developers on their development machines. Later, they go into production. For web applications, this means they get deployed to a server environment and are used by real customers.

Such different environments require different configurations. For example, if the application is about to be debugged, stack traces in the UI (see Debug pages in section 5.6.1) may be welcome, but in production, they present a security issue. In real-life scenarios, debug and production phases often exist in parallel. WAFs therefore support the concept of *Environments* (also called *Profiles*).

Environments allow a *per environment configuration*. That is, depending on whether the applications is about to run in a development or production environment, different configurations automatically apply. This configuration usually addresses the data source to be used, the logging level or exception propagation and can be freely adapted by the developers. Typically, WAFs will support `test`, `development` and `production` modes as environments. Build steps then are performed according to the current mode.

Obviously, the WAF should also support an easy switch between environment modes, for example by means of the project generator.

5.5.3 Dependency Management

For WAFs, it makes sense to delegate some of their tasks to third party APIs that are specialized to provide the respective functionality. These APIs, often themselves concerned as frameworks, need to be integrated into the web application. Integration encompasses both a proper configuration as well as the inclusion of required dependencies. Most of the time, the desired API not only consists of a single library, but defines multiple dependencies on required libraries itself. *Dependency management* refers to the task of dealing with all these dependencies and resolving them. The level of support a WAF offers regarding dependency management may differ.

The first generation of a new application should always include all libraries required for compiling and deploying this project stub. Depending on the used build tool, these dependencies might either be resolved when compiling the project the first time or they already come packaged with the WAF itself.

Quite often, WAFs already include functionality that is not necessarily required by the first project stub. They do so because, from their point of view, a usage is highly probable regarding web application development with this WAF. This may make sense when otherwise the developer would need to deal with dependency management concerns (which depends on the WAF). However, if some of these libraries are not required, this raises the question of how these special dependencies are handled by the WAF. The easiest way is to automatically integrate and configure all of them, irrespective of the particular application. This way, the developer can simply use the respective APIs

without having to worry about unresolved dependencies. However, this also increases the application's size, its complexity (regarding configuration) and its deployment time. Spring Roo takes a more intelligent approach here: It assures that only required libraries become part of the project. If new functionality is used within the application, the additional dependencies are resolved automatically. The other way round, libraries the application no longer depends on, because the developer has removed the related functionality, are deleted automatically. The depicted behavior is guaranteed as long as the respective dependencies are known to Spring Roo. Concerning the ways this knowledge is defined and extended, both Grails and Roo make use of a plug-in mechanism.

5.5.4 Plug-ins

Plug-ins (or add-ons) serve different purposes at once. Most obviously, they integrate and configure additional technologies and services. Whenever a developer is faced with a requirement that is not specific to the application under development, this is where a plug-in could come into play [71]. Quite often, this new requirement might depend on a specific library or framework. Without a plug-in, the developer needs to learn the new framework and, concerning its integration, needs to know how to configure it within the WAF in use. In contrast, adding a new add-on is standardized for the specific WAF, the most difficult step often being the search for an appropriate plug-in. Furthermore, many plug-ins make use of the adapter pattern, simplifying the new API to the functionality that is actually required. This reduces the amount of time to invest into the new framework used in the background. Concerning Grails, this approach may also allow using common Java frameworks in a pure Groovy style. Besides, the design of an add-on even makes sense without making it available to the community: Plug-ins support the modularization of an application.

Both Spring Roo and Grails are WAFs build on a plug-in mechanism. Their approaches here are basically the same: Concerning plug-ins, the architecture distinguishes between *base add-ons* and *third party add-ons*. Base add-ons are out-of-the-box features of the WAF. In Roo, they also provide the metadata other plug-ins can build on, making use of the *Roo core* which provides major services add-ons will "almost always require" [1]. Examples for such base add-ons in Roo are integrations of JPA, JMS, finders, email or logging. Third party add-ons are community-driven and are not bundled with the WAF. They are stored in specific repositories that can be accessed by everyone. Obviously, the developers of a WAF should provide an easy mechanism to access and search in this database. This may be a special website or a special feature integrated into the WAF's generator. The amount of functionality provided by the community through add-ons is probably a key factor for the success of certain WAFs like Ruby on Rails.

At the time of writing, Grails already provided a rich set of add-ons. Roo, in contrast, had only little documentation about the overall topic with very few resources on how to write new add-ons. Accordingly, the number of plug-ins was quite small. However, the developers emphasize the importance of add-ons in the Roo landscape and promise to

update the documentation soon.[38] For the developers of a WAF, a plug-in mechanism is a good way to minimize the download size and avoids dealing with legal issues [1].

Despite all the time saved by using third party add-ons, developers should never forget that plug-ins are external code. For enterprise applications, this means that new plug-ins should be studied in-depth before usage, to find possible security or performance issues.

5.6 Testing

Testing has become an important part in today's software development. Software tests address the reliability of an application and allow finding defects that prevent it from running correctly. Certain kinds of software tests can also be automated, that is, they can be used to verify the application's correctness in a recurrent manner. This is important, for code changes often involve unintended side effects. Using test automation approaches, refactoring existing solutions is possible with a higher confidence [71].

Because of this importance, test-driven development, a software development process that has evolved in recent years, even recommends to "test first", that is, to write tests for a code segment which is yet to be written. This way, a higher test coverage can be achieved and the resulting application is designed with testability from the beginning [71].

Concerning software tests, web applications introduce special requirements. One reason for this is their inherent distributed client/server architecture and the communication over HTTP. Special markup is exchanged between both parties. Furthermore, code may either run in the client's browser or server-side. One also has to deal with the problem of different browsers that may access the web application. Depending on the audience, performance testing may also be important (for further web-related requirements, see [70]). Obviously, these special characteristics also need to be addressed by WAFs.

5.6.1 Manual Testing

Even without writing a single test, developers that deploy their application from time to time and look for the result are also testing [4]. Though such tests aren't automated (hence called *manual* tests), they are nonetheless important and therefore specially supported by WAFs.

Hot Deployment

As already discussed in section 5.5.1, web applications need to get deployed to a server to see them running. For the developer, the recurrent process of compiling, packaging and deploying has a significant drawback: The bigger a web application gets, the more

[38] see `http://forum.springsource.org/showpost.php?p=328167&postcount=15`

increases its startup time on the server. By default, changes to files, irrespective of being plain configuration, resource or source files, always require a complete redeploy. The developer, interested in the effect of the changes applied to the application, has to wait for the application to load. Depending on the frequency of this rhythm of changing and testing, this may lead to a big loss of development time. A recent survey [75] with more than 1000 participants shows that users of different web and application servers spend between 14 to 24 % of their coding time for redeployment.

The idea of an incremental hot deployment is to update an application while it is already running, making changes take effect immediately. This way, it could be a huge time saver; however, Java EE does by no means address this feature [4]. Hot swapping, a feature part of the JVM, allows bytecode changes of loaded classes, but schema changes in these classes fail [28]. The developers of JRebel[39] say their tool is able to deal with most of these schema changes and provide support for many WAFs including Seam and Spring MVC. However, JRebel is only available for a fee.

Due to the missing specification and the inherent complexity of hot deployment, the support of a WAF will usually be home-grown and limited here. At present, a change of persistent entities and EJBs will very likely not be possible. Moreover, the available features here might even depend on the package format of the application (`war` or `ear`) and even worse, on a specific web or application server. In contrast, the change of static resources like style sheets and graphics is often possible. When runtime JSP compilation is supported by the server and is enabled, same holds true for JSP [4]. For the hot deployment of Facelets, they must run in development mode [4]. However, changing a view often requires changing the source code as well. Here, Seam currently supports the change of JavaBeans and Grails the change of controllers and services, though restrictions occur regarding the already mentioned schema changes.

Another question is how a hot deploy can be invoked. Seam offers the `explode` command as an alternative to package deployment which extracts the application to a directory on the server. This way, the replacement of single files becomes possible. The WAF might also support an automatic hot deployment from inside an IDE, e.g., when saving changes to a file.

Obviously, hot deployment for web applications is still an open issue, depending on non-standard features certain web servers might or might not support. To benefit from the solutions a WAF offers in this regard, one often becomes dependent on a specific server vendor and, if IDE support is also desired, maybe also on a specific IDE. Even then, hot deployment of application logic and persistent entities is restricted.

Debug pages

Finding a bug by manual testing is only half the job; it still needs to be corrected. At this point, the distributed nature of web applications and the separation into code and

[39] Project Site: `http://www.zeroturnaround.com/jrebel`

markup becomes obvious. Stack traces, if available at all, tend to be less readable and vague in the way that they often cannot point out the root of a problem.

Debug pages address this issue. During development, they catch any exception that occurs, at a central point. Besides showing the respective stack trace, they also allow getting an overview of the current application state. For example, to locate the cause of the exception, the developer might look for the contents of certain variable scopes or the request parameters sent in the respective request. This way, debug pages offer similar functionalities debuggers provide, in a platform- and IDE-independent manner. Debug pages can also be useful in situations where no exception occurred, to study the current state and other ongoing activities, like sessions or conversations [4].

It should be noted that debug pages are usually only available when the application is running in *debug* or *development mode*. In production, providing such information to the user could easily result in vulnerabilities.

5.6.2 Unit Testing

In contrast to manual tests, unit tests (and all other tests discussed for the rest of this chapter) allow for automation. That is, once written, they can be invoked without human interaction (given that a properly configured testing environment exists). Besides taking work away from the developers, this can also increase an application's quality for errors are found earlier in time.

Concerning dynamically typed languages, like Groovy used in Grails, automated testing is especially important: As already mentioned, errors regarding typing (e.g., class cast exceptions or missing method parameters) cannot be discovered until runtime here. This disadvantage (compared to static typing) can be addressed by extensive testing, and first of all, unit testing. Though this may appear like the time saved by writing dynamic code is lost again for writing tests to verify this code, this does not have to be the case from a test-driven development perspective: For this movement postulates a high code coverage by means of software tests, the tests would have to be written anyway, irrespective of the language under usage [71].

Unit tests focus on only one element (or one unit) of the application at a time, like classes and their methods [38]. For web applications, different kinds of units exist. Quite often, these units are unique to web applications (e.g., controllers or command objects) and therefore, a WAF should provide special support to test them adequately. For example, Grails builds on the popular JUnit[40] framework and provides special `TestCase` extensions for controllers and tag libraries.

Since unit tests try to test code in isolation that most of the time depends on other objects (so called collaborators), developers often need to create *test doubles* to make this possible. To simplify such tasks, Spring MVC provides special Mock Classes and Dependency injection extensions for JUnit here. This way, controller requests can easily be generated manually to verify the controller output afterwards.

[40]Project Site: `http://www.junit.org`

As already stated, unit tests are especially important in WAFs based on dynamic typed languages. Accordingly, Grails, as a typical representative for such a WAF, fosters the creation of unit tests through its project generator: Every time a new controller, entity, tag library or service class is created, a corresponding test class for unit testing the respective artifact is also generated.

In this context, it is also important that WAFs follow a certain convention on where to put test classes and how to define methods as test methods, so that the WAF is able to find and execute them automatically. Such conventions are the reason why WAFs should provide commands for the generation and invocation of different test types. Conventions also allow for the generation of special test reports (for example, in HTML) that give an overview on the whole project's test results, including success rates and the time needed to run all tests. Depending on the level of detail, drilling down to a particular test and examine its results may also be possible.

5.6.3 Integration Testing

Unit tests cannot ensure that different elements of the application work together, that is, whether they integrate correctly. This is where integration tests set their focus. They are "designed to test larger parts of an application" [71].

For the developer, it is important to note that integration tests, from a code-based point-of-view, look quite similar to unit tests. The only difference is that test doubles are no longer required. Therefore, integration tests can for example run against a real database (inside of transactions to be rollbacked afterwards). The actual constraints integration tests are subject to depend on the environment the WAF has set up. Typically, views cannot be tested, but dependency injcetion should already be supported.

The main disadvantage of integration tests is that, due to the special environment required by them, they take much longer to run. Therefore, they cannot provide such an instant feedback unit tests tend to give. In this regard, it should be mentioned that it is not always clear whether to use unit or integration tests to verify certain conditions. The mocking and stubbing required by unit tests often can be too tedious and time-consuming.

As already stated, the special concepts web applications build on introduce special requirements concerning testing. One is the test of proper URL mappings. Here, the developer can verify that a call of particular URL is mapped to the desired controller action and that certain parameters get bound correctly. JBoss Seam introduces a related test mechanism for JSF that allows emulating complete user interactions (but still in the form of integration tests, that is, without the requirement to deploy the application). Here, the interaction between JSF components and the view is tested, without testing the view itself. This basically works by simulating the different JSF lifecycle phases by hand.

5.6.4 Functional Testing

Though integration tests already cover a lot of functionality, some artifacts, especially views, still remain untested. Functional test make sure that "the application works as a whole" [71]. For this, the application runs in its common environment, that is, an unrestricted web or application server.

The last condition already reveals a significant drawback of functional tests: For they run in an unrestricted environment, a single test demands much more time than for example a unit test. However, since functional tests also aim for automation, this is usually no time a developer has to invest.

Another drawback, possibly more important regarding productivity, relates to the fact that functional tests tend to be UI-driven. The fact that user interfaces are often subject to frequent changes, makes functional tests very fragile. Their automation therefore often requires frequent modifications.

There exist different kinds of functional tests. WAFs should at least integrate one mechanism to test views and their interaction with controller code. Plug-ins might then provide additional features to the developer. The *Functional Test* plug-in for Grails is a typical example. It allows communicating with the server over HTTP and verify different things about the response, such as status codes or page contents.

Selenium[41] is another tool WAFs often integrate support for. The tool makes use of a common web browser to test interactions with the web application. The most important advantage is that Selenium tests are easy to create using point-and-click interfaces provided for certain Web browsers. However, tests similar to the ones created by the Functional Test plug-in are likely less UI-dependent. Selenium in turn also allows testing for correct JavaScript handling, another thing that can be addressed using functional tests.

[41] Project Site: `http://seleniumhq.org`

Chapter 6

Conclusion

This chapter first recapitulates the work and discusses its central results (section 6.1). Those interested in a complete overview of the productivity criteria should refer to Appendix A. Section 6.2 takes a look at the future trends to expect regarding Web Application Frameworks and their way of use. In the last section, open research questions are discussed.

6.1 Results

The last chapter has discussed the productivity criteria of Web Application Frameworks in detail. To provide a quick overview, Appendix A lists all criteria in a comprehensive manner. This list encompasses more than 120 entries.

As this huge amount of criteria indicates, there already exists a large support for Web application development through WAFs: WAFs can automatically configure appropriate environments for testing, development or production phases. They are able to reverse engineer the database to convey the domain model into the required entity representation. Validation configuration is executed without further ado prepared to be used in declarative style, with separation of concerns in mind. Dynamic finders can easily be generated, either at compile time or runtime; basic queries no longer have to be written manually. Views for specific entities or query result sets can be created automatically, together with features allowing for pagination and sorting to foster usability. The Views adhere to composition principles to support reuse on markup level. Internationalization is easy to integrate. Context scopes manage the application state automatically and can be accessed easily. Specific business requirements often come already shipped in form of plug-ins. These are only the most basic features modern WAFs provide. The list does not even contain the general concepts provided to ease work, like special IDE support, the rich set of features that come with the project generator or the ways to ease testing.

One noticeable fact is the extent of technical evolution regarding the former act of

Servlet programming, even with respect to the enhancements in the latest Servlet specification (see section 2.4.2). Especially the WAFs in the area of request-based frameworks (which have a strong relationship to Servlets, see section 2.5.3), as for example Spring MVC, show that the same programming model by now allows for code that feels quite natural to Java developers. This for example reflects in the way input can be validated and passed to the application. Method bodies are relieved from validation code and no longer depend on the excessive API of Java's `ServletRequest` or `ServletResponse` objects. Parameters in the method's signature can be defined very freely, based on Java's Reflection mechanism. The difference between classes serving normal objects and those serving web clients fades way. Obviously, this reduces the effort of implementation and later rework.

Such simplifications in the programming model are even pushed further by Domain Specific Languages, easy to implement with dynamic languages like Groovy. One place Grails makes use of a this concept is its solution for Object-relational mapping, GORM (see section 4.2.3). Domain Specific Languages focus on particular domains, this way hiding unnecessary complexity from the programmer. However, the simplicity and productivity of a certain DSL in this case depends on the programming language used for its implementation.

Though Domain Specific Languages are a good example for the power of dynamic languages, it is not so clear whether there really is a productivity gain emanating from them. Their combination with dynamic typing, their support of special operators, and the often-to-find influences from the functional programming paradigm generally seem to make code written in dynamic languages shorter. However, type-related errors aren't found until runtime here, which can be a huge disadvantage especially for web applications and their long-taking deployment times. Though this issue can be partly addressed by excessively writing tests (which is also a good practice even for static languages) and there even exist special IDE plug-ins that try to bring code-completion features to dynamic programming, this behavior may easily compensate any productivity advantages here. Furthermore, as the last chapter shows, both JBoss Seam and Spring MVC offer ways to implement code written in dynamic languages for the JVM. Even though they lack a DSL support, this can be useful in those situations where static languages aren't the best choice. Nevertheless, this influence of dynamic languages on productivity in web development is definitely worth investigation.

None of the examined WAFs scores on every single criterion. One artifact that discloses significant differences is the project generator. All solutions had a specific handling here and were of varying effectiveness. The Roo shell was intuitive in use and proved the most powerful tool in the examination. Its code generation support, ranging from big artifacts to single entity fields, was even able to compensate some of the disadvantages inherent to static languages. Furthermore, the shell is designed for a usage throughout the whole development cycle of an application and in an arbitrary amount of projects in parallel. In contrast, JBoss Seam's generator is linked to a specific project at

any time, and whenever a new project is created using seam-gen, this link is broken up forever. A parallel use in different projects therefore is not possible.

A second feature that may differ between frameworks and comes with productivity implications is the generation of finders, that is, database queries. Grails supports finders leveraging Groovy's `MetaClass` and `methodMissing` features, that is, they are created dynamically (hence called *dynamic finders*). Spring Roo also supports the generation of queries based on specific persistent entities, however, this requires the developer to issue some commands using the shell first. Though this appears slightly more time-consuming, the advantage of this approach is that the resulting queries become part of the static application code – they can be accessed by the developer and adjusted to the specific needs of the use case. In Grails, by contrast, dynamic finders are useless if they do not match the requirements exactly. Despite Spring Roo's powerful approach here, finders are much more difficult to implement using static languages like Java and only using static code generation mechanisms. Seam for example lacks this feature completely.

Regarding productivity, another important aspect with differing implementations is the concept of context scopes. Context scopes are required to overcome the statelessness of the WWW's underlying HTTP protocol. WAF's should at least offer an additional *flash scope*, that is, a scope for storing references that need to survive a redirect. Though often required, such behavior is not part of the Servlet specification. This hinders the implementation of the commonly used Redirect-After-Post pattern. In this case, developers either need to define a custom scope, or store the respective references in the session. The first approach is obviously quite complicated, whereas the latter can easily result in performance overheads and requires a very thorough style of programming, for the developer needs to clean up the session manually. Spring MVC currently offers no appropriate solution in this context. In contrast, Grails again benefits from Groovy's power in domain specific languages and transparently injects suitable containers for all required context scopes.

Since many companies today use IDEs, the availability of respective plug-ins that support the development with a specific WAF, also influences productivity. This is also a point where the popularity of a WAF and the background of its development team may play a role: At least three IDEs with a significant market share exist (see section 5.4.2). As plug-in development for them is not possible in a platform-independent manner, it introduces a huge effort to the WAF developers. The higher the popularity of a framework is, the higher are the chances to find people willing to help. Likewise, if the WAF is developed by a company, more financial resources are available for specific IDE support. Since IDE plug-ins can be very useful, these (non-technical) issues may also be worth considering.

Lots of other criteria that influence the productivity in web engineering are listed in Appendix A. Even though their contribution to productivity seems to be quite small for many of them, in their entirety they provide a very rapid style of programming. However,

this advantage does not come for free: Developers need to actively study the WAF and its possibilities. Modern WAFs comprise multiple artifacts, the project generator being one example, IDE plug-ins another. This can be quite unnatural for developers only used to program against certain APIs. The API that comes with a WAF is still important, but without delving into the details of the other artifacts, teams will not leverage from the full power offered by the respective WAF. They risk to be satisfied with the most prominent features and disregard the rest. So to say, the development with a WAF requires a new style of programming, where switching between many different tools, which may even interfere with each other, needs to be learned first.

When studying the different WAFs for this master thesis, it became obvious that, sometimes, speed of development and quality of the resulting solution directly interfere with each other - development speed reduces the input, quality increases the output. The productivity of this particular solution depends on the team's valuation of each factor. Sometimes, the decision is even a hard-wired part of the WAF and there is no way to adapt the respective behavior. One framework might decide for a higher development speed, the other for quality. This is where the framework selection can become quite tricky or may turn out as a bad decision if a major aspect of the project is affected. To illustrate this, there for example is a significant difference in the way Grails and Seam handle the persistence context in their ORM solutions. Grails and its Domain Specific Language GORM completely hide it from the developer; there is no way to access it. Instead, the persistence context is automatically scoped to the current transaction. That means if one transaction stores an entity in the context, and a subsequent one tries to access the now detached data, it has to be merged with the current database first. Depending on the amount of data that has to be merged, this can be a very expensive operation (the details are described in section 5.3.5). Seam, in contrast, offers a way to set the persistence context's scope manually. This prevents performance leaks, but comes to price of a more complicated Application Programming Interface (API). Many concepts in Grails, like for example dynamic finders, would not be possible with such an implementation.

One aspect with a huge need for improvement is hot deployment. A recent survey shows that developers estimate to spend up to 24 % of their coding time for the deployment of their applications. None of the examined frameworks was able to deploy classes with schema changes while the application was already running. There is one tool in the market, available for a fee, that seems to be able to deal with such schema changes. This indicates that solutions do exist, but are obviously hard to implement. Hence, this may be an area that should best be addressed by Java EE itself and cannot be delegated to the framework developers. A fully working approach would be a big step forward in web engineering productivity.

6.2 Outlook

The current state-of-the-art in WAF development already allows developers to set their main focus on the application's business code (and markup). However, on the side of the framework, this requires implementing the depicted criteria in their entirety. The fact that none of the examined frameworks was capable of that implies that such a WAF is probably not available at the moment. Hence, the first step towards a higher productivity in web engineering is by adapting the full range of criteria into the WAF. Regarding the level of competition in the market, and the huge productivity improvements in recent years, an alignment is likely to happen.

In the future, WAFs will probably need to expand their capabilities to also assist the developer in writing the core business aspects of the application. Spring Roo already provides a mechanism that foreshadows how such assistance may look like. Here, each action of the developer is tracked by a tool in the background, reacting on certain events by modifying the application. The roundtrip-awareness of this tool, together with a separation of generated and hand-written code, together avoid an interference with the developer. Such a mechanism is especially appealing for web application development since the developer has to deal with lots of different artifacts here. For the application flow always follows a fixed sequence of steps here, writing the respective tools should not be too complicated. However, as already stated in the last section, such tools confront the developer with a new style of programming: At the moment, programmers aren't used to environments that change the code base by themselves.

Another aspect that is gaining more and more importance is integration. Integration issues are probably among the most time-wasting tasks a web developer has to fulfill. A certain library or framework needs to be accessed by the application; however, neither the WAF developers nor the developers of this library themselves did anticipate this usage scenario. In the worst case, this may even lead to a situation where a proper integration is not possible. Besides being obviously unsatisfying, it is also difficult to foresee and hence, may even let the whole project fail. It is quite possible that, when technical features of WAFs align more and more in the near future, the integration capabilities of a WAF become the most important factor to choose among them.

One way some WAFs try to address integration support today is by plug-in systems. The idea behind such a system is that the corresponding work is done once by a third party and can be reused by others. However, the quality of the solutions will obviously vary from plug-in to plug-in this way. Of the examined WAFs, the only one without an integrated plug-in system was JBoss Seam. Regarding Seam 3, a CDI Plug-in-Center already exists.[42]

[42] see `http://groups.diigo.com/group/cdi-extensions`

For some parts of a web application, visual editors, integrated into an IDE, may also be helpful. One typical example for this is the JPA Diagram Editor.[43] This tool provides a visual editor to create an application's domain model. However, using such tools today narrows the developer to a specific IDE: Providing visual editors is not only costly, it also requires much rework to offer the same tool for another IDE. In the future, such tool support will likely exist mostly for those technologies that are based on a standard – like in the above example, JPA. Standardization is a precondition for many IDE-specific development simplifications, for otherwise, the demand may not justify the development. Page flows are only one example that seems predestinated for visual tool support. Developers could first define the flow in a UML-like language and then generate the required code, view and configuration files. However, page flows aren't standardized yet and such a tool does not exist.

6.3 Future Work

This master thesis worked out the technical criteria that define a WAF's productivity. As a comparison with the related concept of productivity factors in software development projects shows, it is also a common approach to measure the impact of such factors. Due to the lack of a decent project database (see section 3.4), in the course of this work, this was not possible though. The creation of such a database would allow a more detailed examination of the factors that distinguishes web development from common software development. When measuring the impact of certain technical WAF criteria, one can fall back to the results worked out here. To this, the measurement should obviously focus on the likely more crucial criteria than on all of them in their entire detailedness.

One point outside the scope of this work but probably worth investigation are special business requirements in Web development. Here, the term business requirements refers to all those features provided by some WAFs but cannot be concerned a part of their core functionality. Typical examples are full-text search, charting or PDF generation. Though not used by every single web project, these are still common requirements. Hence, an expansion of the criteria catalog by taking business requirements into account may provide helpful.

Even with the productivity criteria at hand, framework selection can still be difficult. Here, development teams would clearly benefit from a detailed instruction on how the process should be designed best. Often, the time to select a WAF is quite short and leads to wrong decisions. The process therefore should take different time spans into account and adapt the required steps accordingly. However shaped, it would probably contain one or more phases in which certain WAFs have to be explored in some way. This is also the point where applying the criteria could become a fix part of the process, for such an examination will also accelerate the understanding of the respective WAF.

[43] see `http://wiki.eclipse.org/JPA_Diagram_Editor_Project`

The last paragraph also indicates another way the criteria can be used. By applying the catalog to multiple WAFs, and publishing the respective results, the capabilities of different frameworks could be compared in an objective manner. This would obviously simplify the framework selection in many development teams, because they no longer need to apply the catalog themselves. The only task left would be to weight the criteria in an order that reflects the project's requirements.

Bibliography

[1] Ben Alex. Spring Roo 1.0.0: Technical Deep Dive. Presentation on SpringOne 2GX, 2010.

[2] Ben Alex. What's New In Spring Roo 1.1.1. `http://blog.springsource.com/2011/01/11/whats-new-in-spring-roo-1-1-1/`, January 2011.

[3] Ben Alex and Stefan Schmidt. Spring Roo – Reference Documentation. `http://static.springsource.org/spring-roo/reference/pdf/spring-roo-docs.pdf`, May 2010.

[4] Dan Allen. *Seam in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[5] Helmut Balzert. *Lehrbuch der Softwaretechnik*. Spektrum, Akad. Verl., Heidelberg, 1998.

[6] Bryan Basham, Kathy Sierra, and Bert Bates. *Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam*. O'Reilly Media, second edition edition, 3 2008.

[7] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[8] Emmanuel Bernard and Steve Peterson. JavaServer Faces Specification - Version 2.0. `http://jcp.org/aboutJava/communityprocess/final/jsr314`, June 2009.

[9] Emmanuel Bernard and Steve Peterson. JSR 303: Bean Validation. `http://people.redhat.com/~ebernard/validation/`, October 2009.

[10] Barry Boehm. Managing software productivity and reuse. *Computer, IEEE*, 32:111 – 113, 1999.

[11] Barry W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, 1987.

[12] Barry W. Boehm. Safe and simple software cost analysis. *Software, IEEE*, 17(5):14–17, 2000.

[13] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald Reifer, and Bert Steece. *Software Cost Estimation with Cocomo II*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2000.

[14] Tilmann Bruckhaus, Nazim H. Madhavji, Ingrid Janssen, and John Henshaw. The impact of tools on software productivity. *Software, IEEE*, 13:29, 1996.

[15] Julian M Bucknall. 10 mistakes every programmer makes. `http://www.techradar.com/.../10-mistakes-every-programmer-makes-909424`, November 2010.

[16] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[17] Sven Casteleyn, Florian Daniel, Peter Dolog, and Maristella Matera. *Engineering Web Applications (Data-Centric Systems and Applications)*. Springer, 1 edition, 8 2009.

[18] Nicholas Chen. Convention over Configuration. `http://softwareengineering.vazexqi.com/files/pattern.html`, November 2006.

[19] Roberto Chinnici and Bill Shannon. Java Platform, Enterprise Edition (Java EE) Specification, v6. `http://www.jcp.org/en/jsr/detail?id=316`, December 2009.

[20] Scott Davis. Mastering Grails: Understanding plug-ins. Mix in new functionality with ease. IBM developerWorks, `http://www.ibm.com/developerworks/java/library/j-grails07219/index.html`, July 2009.

[21] Gibeon Soares de Aquino Junior and Silvio Romero de Lemos Meira. Towards effective productivity measurement in software projects. *Software Engineering Advances, International Conference on*, 0:241–249, 2009.

[22] Suzana Candido de Barros Sampaio, Emanuella Aleixo Barros, Gibeon Soares de Aquino Junior, Mauro Jose Carlos e Silva, and Silvio Romero de Lemos Meira. A review of productivity factors and strategies on software development. *Software Engineering Advances, International Conference on*, pages 196–204, 2010.

[23] Fergal Dearle. Metaprogramming and the Groovy mop. http://www.packtpub.com/article/metaprogramming-and-groovy-mop, May 2010.

[24] Dániel Dékány. Freemarker Manual. http://freemarker.sourceforge.net/docs/index.html, December 2009.

[25] T. DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[26] Linda DeMichiel. Java Persistence api, version 2.0. http://jcp.org/aboutJava/communityprocess/final/jsr317, December 2009.

[27] Gordon Dickens and Ken Rimple. Spring Roo in Action. Book accessed through the Early Access Program of Manning (MEAP), print edition planned to be published in Spring 2011, December 2010.

[28] Stuart Douglas. Current state of hot deployment in java. http://community.jboss.org/wiki/Currentstateofhotdeploymentinjava, June 2010.

[29] Dirk Draheim and Gerald Weber. An overview of state-of-the-art architectures for active web sites. Technical report, Free University Berlin, 2002.

[30] Khaled El Emam and A. Günes Koru. A replicated survey of it software project failures. *IEEE Software*, 25:84–90, 2008.

[31] Jason Farrell and George S. Nezlek. Rich internet applications - the next stage of application development. In *ITI '07: Information Technology Interfaces,*, pages 413 – 418, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[33] Anatoly Fedosik, Olga Chikvina, Michael Sorokin, and Svetlana Mukhina. Seam Development Tools Reference Guide. http://docs.jboss.org/tools/3.1.0.GA/en/seam/html/index.html, 2010.

[34] Seth Fogie. How Not to Use Cookies. http://www.informit.com/guides/content.aspx?g=security&seqNum=232, December 2006.

[35] Apache Software Foundation. Apache Tiles Tutorial. http://tiles.apache.org/framework/tutorial/index.html, June 2010.

[36] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[37] Martin Fowler. InversionOfControl. `http://martinfowler.com/bliki/InversionOfControl.html`, June 2005.

[38] Martin Fowler. Mocks Aren't Stubs. `http://martinfowler.com/articles/mocksArentStubs.html`, January 2007.

[39] Martin Fowler. Using Domain Specific Languages. `http://martinfowler.com/dslwip/UsingDsls.html#DefiningDomainSpecificLanguages`, April 2008.

[40] George Franciscus and Craig R. McClanahan. *Struts in Action: Building Web Applications With the Leading Java Framework*. Manning Publications Co., Greenwich, CT, USA, 2002.

[41] Tom Gilb. *Software metrics (Winthrop computer systems series)*. Winthrop Publishers, 1977.

[42] Andreas Grabner. Top 10 Performance Problems taken from Zappos, Monster, Thomson and Co. `http://www.theserverside.com/discussions/thread.tss?thread_id=60382`, June 2010.

[43] Robert Hanson and Adam Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications, 6 2007.

[44] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer, IEEE*, 34:120, 2001.

[45] Jacob Hookom. Facelets - JavaServer Faces View Definition Framework. `https://facelets.dev.java.net/nonav/docs/dev/docbook.html`.

[46] José Ignacio Fernández-Villamor, Laura Díaz-Casillas, and Carlos Á. Iglesias. A Comparison Model for Agile Web Frameworks. In *EATIS '08: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[47] Eric Jendrock, Ian Evans, Devika Gollapudi, Kim Haase, and Chinmayee Srivathsa. The Java EE 6 Tutorial. `http://docs.jboss.org/weld/reference/1.0.1-Final/en-US`, November 2010.

[48] Brian Johnson, Ralph E. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[49] Michael Jouravlev. Redirect After Post. `http://www.theserverside.com/news/1365146/Redirect-After-Post`, August 2004.

[50] Gerti Kappel, Birgit Pröll, Siegfried Reich, and Werner Retschitzegger, editors. *Web Engineering - The Discipline of Systematic Development of Web Applications*. John Wiley & Sons Ltd., England, 2006.

[51] Gavin King, Pete Muir, Dan Allen, and David Allen. JSR-299: The new Java standard for dependency injection and contextual lifecycle management. `http://download.oracle.com/javaee/6/tutorial/doc`.

[52] Gavin King, Pete Muir, et al. Seam – Contextual Components, A Framework for Enterprise Java. `http://docs.jboss.org/seam/2.2.0.GA/en-US/pdf/seam_reference.pdf`, July 2009.

[53] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[54] Holger Koschek, Oliver Ihns, Dierk Harbeck, Stefan M. Heldt, Jo Ehm, Carsten Sahling, and Roman Schlömmer. *EJB 3 professionell*. Dpunkt.verlag Gmbh, 2007.

[55] Cameron McKenzie. Moving from Spring to Java EE 6: The Age of Frameworks is Over. `http://www.theserverside.com/discussions/thread.tss?thread_id=61023`, October 2010.

[56] Bernd Müller and Harald Wehr. *Java-Persistence-API mit Hibernate*. Addison-Wesley, Munich, Germany, 2007.

[57] Vu Nguyen, LiGuo Huang, and Barry Boehm. An analysis of trends in productivity and cost drivers over years. Technical report, Center for Systems and Software Engineering, University of Southern California, August 2010.

[58] Jeff Offutt. Quality Attributes of Web Software Applications. *IEEE Softw.*, 19(2):25–32, 2002.

[59] Ed Ort. New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine. `http://java.sun.com/developer/technicalArticles/DynTypeLang/index.html`, July 2009.

[60] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40:12–15, February 2007.

[61] Christy Pettey and Ben Tudor. Gartner Says Worldwide enterprise Software Revenue to Surpass $232 Billion in 2010. `http://www.gartner.com/it/page.jsp?id=1437613`, September 2010.

[62] Matt Raible. Comparing Java Web Frameworks. Presentation on ApacheCon, 2007.

[63] Matt Raible. Comparing JVM Web Frameworks. Presentation on Devoxx, 2010.

[64] Graeme Rocher and Jeff Brown. *Grails 1.2. Das produktive Web-Framework für die Java-Plattform*. Mitp-Verlag, Bonn, Germany, February 2010.

[65] Graeme Rocher, Peter Ledbrook, Marc Palmer, and Jeff Brown. The Grails Framework – Reference Documentation. `http://grails.org/doc/latest/guide/single.pdf`, June 2010.

[66] Melanie Ruhe, Ross Jeffery, and Isabella Wieczorek. Cost estimation for web applications. *Software Engineering, International Conference on*, 0:285, 2003.

[67] Shin Sang. How to Choose A Web Application Framework. Presentation on Hong Kong Hospital Authority, 2007.

[68] Govind Seshadri. Understanding JavaServer Pages Model 2 architecture. `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html`, December 1999.

[69] Tony C. Shan and Winnie W. Hua. Taxonomy of Java Web Application Frameworks. *E-Business Engineering, IEEE International Conference on*, pages 378–385, 2006.

[70] Vijay Shinde. How can a Web site be tested? `http://www.softwaretestinghelp.com/how-can-a-web-site-be-tested`, July 2007.

[71] Glen Smith and Peter Ledbrook. *Grails in Action*. Manning Publications Co., Greenwich, CT, USA, 2009.

[72] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[73] Iwan Vosloo and Derrick G. Kourie. Server-Centric Web Frameworks: An Overview. *ACM Comput. Surv.*, 40(2):1–33, 2008.

[74] Eberhard Wolff. *Spring 3: Framework für die Java-Entwicklung*. dpunkt Verlag, Heidelberg, Germany, 3. edition, February 2010.

[75] ZeroTurnaround. What happens when over 1000 Java developers compare their development environments? `http://www.zeroturnaround.com/java-ee-productivity-report-2011`, December 2010.

Appendix A

Catalog of Criteria

The following table gives an overview of the productivity criteria of Web Application Frameworks worked out in chapter 5. The catalog is sorted according to the different sections. Each criterion is tagged with a unique id; for each section, numeration starts again, combined with a single letter that identifies the respective section. This approach should simplify the work with the catalog and ease the discussion about single criteria. Furthermore, a short explanation is given for each point; in case more detailed information is required, a reference to the section or page discussing the specific criterion is provided.

The table also includes an estimation regarding a criterion's influence on the developer's productivity, ranging from `LOW` to `HIGH` in three levels. However, this also depends on the respective project.

A.1 Model (M)

No.	Criterion	Description	Ref.	Influence
Configuring the Database Connection				
M1	Project Generator Support	Instead of manually defining the connection settings, the WAF's generator provides support here. Configuration files are created automatically and put to the right place inside the project. Reasonable default settings are used where possible, following a Convention over Configuration approach.	5.1.1	LOW
M2	Support of particular Vendors	The WAF provides support for specific, often used database vendors. Hence, the configuration support can be improved by making the JDBC driver available automatically and introducing vendor specific configurations.	5.1.1	LOW
M3	Reconfiguration Support	The WAF's support regarding database configuration is available throughout the whole development phase, not only when creating a new application.	5.1.1	LOW
M4	Environment Configuration	The WAF's support regarding database configuration is reusable for different environments, like test, development or production profiles.	5.1.1	LOW
Reverse Engineering the Database				
M5	Reverse Engineering Support	The WAF is able to introspect a database and generate persistent entities on the basis of this information. The importance of this feature also depends on how close the model code used by the WAF sticks to the standard. If standard, a third party tool tailored for reverse engineering will likely exist elsewhere.	5.1.2	HIGH
M6	Incremental Reverse Engineering	If the database is accessed and modified by other developers or applications during development, incremental reverse engineering helps to discover schema changes and adapt the application's model automatically, if desired.	5.1.2	LOW

No.	Criterion	Description	Ref.	Influence
M7	Simple Reverse Engineering Control	A simple reverse engineering control allows including or excluding certain tables from the reverse engineering process. Besides accelerating the reengineering process, this relieves the developer from deleting undesired entities, references to these objects, and any artifacts created in conjunction with the entities.	5.1.2	LOW
M8	Custom Reverse Engineering Strategies	Custom strategies make it possible to address undesired naming conventions, customize type mappings or provide additional information the reverse engineering process cannot gather from the metadata stored in the database alone.	5.1.2	AVERAGE
Top-down Development Support				
M9	Stub Generation	The WAF's generator provides a way to generate stubs for persistent entities. Those stubs are properly configured within the framework, but contain no application-specific code yet. The stubs may already define `id` and `version` fields.	5.1.3	AVERAGE
M10	Field Generation	If the generation of single persistent entities is possible, the WAF might also support the definition of fields and relationships to other entities by means of its project generator.	5.1.3	LOW
M11	Accessor Generation	For every newly defined field, the WAF automatically generates the respective getter and setter methods. Though often required by various frameworks, this is something easily forgotten by the developer – a mistake not discovered until deployment.	5.1.3	AVERAGE
M12	Template Adaption	To generate entities, WAFs typically make use of special template engines. The underlying templates should be accessible to the developer for adjusting the code generation process to the conventions used in the respective development project.	5.1.3	LOW

No.	Criterion	Description	Ref.	Influence
Validation of Entities				
M13	Separation of Concerns	The WAF provides an explicit validation support aiming for separation of concerns. One possible approach is to provide an implementation of the Bean Validation specification.	5.1.4	AVERAGE
M14	Declarative Approach	Validations are defined in a declarative manner. This prevents the developer from stringing together `if` conditions and other assertions. Such approaches are too error-prone and time-consuming in comparison.	5.1.4	HIGH
M15	Reverse Engineering	The information available from the database and JDBC metadata is used by the reverse engineering process to declare validation constraints for fields or whole classes.	5.1.4	AVERAGE
M16	Cross-field Validations	The declaration of constraints concerning multiple fields is possible in a way consistent to the WAF's overall validation approach.	5.1.4	LOW
M17	Generator Support	If the WAF supports the definition of single entity fields through the generator, then the declaration of field-specific validations is possible in the same way.	5.1.4	LOW
Entity Lifecycle Management				
M18	Persistence Context Hiding	Simple operations on entities are supported by hiding the persistence context, e.g., by using an Active Record pattern approach. This relieves the developer from obtaining a persistence context and handle it explicitly, unnecessary for simple use cases.	5.1.5	AVERAGE
M19	Extended Persistence Context Handling	The WAF explicitly supports the storage of single entities in long running conversations and extended persistence contexts. Otherwise, costly merge operations are required. Implementing an own solution here is a complex task.	5.1.5	AVERAGE

No.	Criterion	Description	Ref.	Influence
Database Queries				
M20	Query Support	The WAF provides support for handling result sets in long running conversations and extended persistence contexts. Otherwise, costly and possibly time-consuming merge operations are required.	5.1.6	HIGH
M21	Dynamic Finders	By examining an entities structure, simple queries required by the developer can be generated automatically, using features provided by dynamic languages.	5.1.6	HIGH
M22	Static Finders	By examining an entities structure, simple queries required by the developer can be generated automatically and become part of the application code. This way, it becomes possible to start with a generated stub and reuse it for own modifications.	5.1.6	HIGH
M23	Finders: Operator Support	The fields of the entity can be queried in multiple ways, e.g., using LIKE, EQUAL or BETWEEN operators.	5.1.6	AVERAGE
M24	Finders: Depth	Finders are able to incorporate multiple fields – best, an infinite amount of fields, combining them using AND or OR operators.	5.1.6	AVERAGE

A.2 View (V)

No.	Criterion	Description	Ref.	Influence
CRUD Pages				
V1	Create, Read, Update, Delete, List	The WAF provides support for generating the most basic CRUD pages, that is, views to create, show, edit or delete single entities. These operations (except Create) also imply a possibility to list existing entities in a first step. Such a list then allows drilling-down to specific entities to read, update or delete them.	p. 65	HIGH

No.	Criterion	Description	Ref.	Influence
V2	Create, Update: Validation / Type consideration	When generating the markup, scaffolding takes the type of entity fields and the corresponding validation restrictions into account. This results in adequate input fields with explicit depiction of required values. To support validation, the WAF might also go an AJAX-based approach here.	p. 65	HIGH
V3	Create, Update: Separate Error fields.	For providing a better usability in input form, error messages (resulting from validation processing) are separated into multiple fields, each one related to a single entity field.	p. 65	HIGH
V4	Searching	The WAF supports to search for entities of specific types. By analyzing the defined fields, this for example allows using a Query by Example approach for Search. Concerning the result pages, the same requirements as for Lists exist, that is, they should allow drilling-down to particular entities to show, update or delete them.	p. 65	HIGH
V5	List, Searching: Sorting	When supported, the lists and result sets can be sorted by different criteria, depending on the fields defined in the respective entity, in an ascending or descending manner, as the user desires.	p. 65	AVERAGE
V6	List, Searching: Pagination	When supported, the lists and result sets can be truncated into multiple pages. This allows for a better clarity in the view and reduces the database load. The user might also be able to define how many entities show up on a single page.	p. 65	HIGH
V7	Finders	If the WAF is able to generate static finders, the generation of adequate search screens that invoke these finders becomes possible.	p. 65	HIGH
V8	I18n Configuration	The generated CRUD screens should be fully configured regarding internationalization concerns.	p. 65	HIGH

No.	Criterion	Description	Ref.	Influence
Dynamic Scaffolding				
V9	Complete Generation	The most basic support for dynamic scaffolding allows generating all available views for all available entities automatically.	p. 67	AVERAGE
V10	Specific Entity Types	The WAF allows applying its dynamic scaffolding approach with respect to specific entity types. The functionality regarding different types can be provided explicitly by the developer.	p. 67	HIGH
V11	Specific CRUD-Pages	The WAF makes it possible to define which parts of the available CRUD screens should be generated dynamically and allows defining other functionality manually.	p. 67	HIGH
V12	Influence on generated Pages	The WAF allows controlling the dynamic scaffolds to a certain degree, e.g., the ordering of input fields.	p. 67	LOW
Static Scaffolding				
V13	Complete Generation	The most basic support for static scaffolding allows generating all available views for all available entities automatically.	p. 67	HIGH
V14	Specific Entity Types	The WAF allows applying its static scaffolding approach with respect to specific entity types. This is possible at any time during development.	p. 67	HIGH
V15	Specific CRUD-Pages	The WAF makes it possible to define which parts of the available CRUD screens should be generated. When only specific CRUD screens are required, this feature avoids the deletion of unused pages. It remains available throughout the whole development process.	p. 67	LOW
V16	Round-tripping	Scaffolded pages can be modified by the developer while keeping view generation running (as in dynamic scaffolding), but without interfering with the changes made. Changes in the underlying entities are automatically propagated to all relevant views.	p. 68	HIGH

No.	Criterion	Description	Ref.	Influence
Template Adaption				
V17	Adaption of global View Templates	To prevent applying the same changes required for any new web application over and over again, the WAF allows adapting the globally applied view templates used for CRUD generation.	p. 69	AVERAGE
V18	Adaption of local View Templates	To prevent applying the same changes required for any new page of the web application over and over again, the WAF allows installing the global templates into single web applications. From there on, local templates are used for the generation of CRUD pages. This way, local template modifications become possible.	p. 69	HIGH
View Composition				
V19	Explicit Composition Support	The WAF supports view composition independent of a specific view technology, e.g., by implementing the Decorator or Composite View pattern.	5.2.2	HIGH
V20	View Technology supports Composition	If explicit composition support is not available, at least the used view technology should support composition then. However, such solutions generally cannot keep up with explicit composition support.	5.2.2	HIGH
Internationalization				
V21	Pre-configuration	All configuration steps, including the creation and positioning of resource bundles for the default locale, are automatically carried out by the WAF.	5.2.3	AVERAGE
V22	Automatic Locale Detection	The WAF integrates a mechanism to detect the user's locale and automatically retrieves the corresponding resource bundle.	5.2.3	HIGH
V23	Resource bundle Abstraction	For many libraries and frameworks integrated into a WAF often provide their own i18n-support and come with different resource bundles, the WAF abstracts from this behavior by providing a special component that allows a central access to all these bundles.	5.2.3	AVERAGE

No.	Criterion	Description	Ref.	Influence
V24	Locale Switching	When providing different locales, the WAF eases their testing by simplifying the switch between different languages, e.g., by using special URL parameters that allow a manual definition of the locale to use.	5.2.3	LOW
Tag Libraries				
V25	Availability of Tag Libraries	To avoid the implementation of new tags for the application, a high amount of (third party) libraries is available for the default view technology used by the WAF.	5.2.4	HIGH
V26	Tag Libraries by the WAF	If the WAF provides certain "sweet spots" in its usage, and they also address the View, then special tag libraries supporting these use cases should come bundled with the framework.	5.2.4	AVERAGE

A.3 Controller (C)

No.	Criterion	Description	Ref.	Influence
Scaffolding				
C1	Complete Generation	The most basic option for a WAF providing controller scaffolding. Controllers for each entity are created, containing all available action methods for all available CRUD pages. These controllers work out-of-the-box, that is, they are fully configured and completely implemented.	5.3.1	HIGH
C2	Specific entity	By pointing the WAF to a single entity, a controller for this specific entity is generated, containing all available controller actions. If the WAF does not provide such an option, this would reduce scaffolding to be used in a reverse-engineering context.	5.3.1	HIGH

No.	Criterion	Description	Ref.	Influence
C3	Specific controller actions	Instead of generating all actions, the WAF provides a way to specify which action methods to implement for the respective entity. Without this option, the additional overhead usually comes down to the removal of the obsolete methods and their corresponding views.	5.3.1	LOW
C4	Scaffolding without View	The WAF provides a way to generate a single controller without generating the associated view(s). Again, this can be regarded less important, for the task handed over to the developer usually is to delete some markup files.	5.3.1	LOW
C5	Static Finders	If the WAF allows generating code for static finders, then this option offers a way to generate controller actions which invoke these finders and return the corresponding result sets to the view.	5.3.1	HIGH
C6	Adaption of templates	For not applying the same changes for every controller over and over again, the controller templates used in the background are open for adaption. This becomes especially important when the development team decides to deviate from the default controller mapping approach.	5.3.1	AVERAGE
Handler Mapping				
C7	Convention	There exists an easy to use default approach for mapping incoming requests to controllers. The use of configuration files is not required, but may be provided as an alternative.	5.3.2	AVERAGE
C8	Strategy switch	It is possible to change the default URL mapping approach without interfering with the existing controller actions. These action methods either remain untouched or are translated according to the new mapping approach automatically.	5.3.2	LOW

No.	Criterion	Description	Ref.	Influence
Data Binding				
C9	*Command Object* support	To ease the handling of input, the WAF supports command objects encapsulating it in an object-oriented way. Command Objects also provide a way to deal with situations where no 1:1 relationship between entities and input data exists.	p. 74	HIGH
C10	*Implicit Object* support	A WAF based on a dynamic language transparently injects (implicit) objects into each controller. Concerning the developer, no further effort is required to access these objects. This also keeps the code clean from technical concerns, being easier to maintain.	p. 75	AVERAGE
C11	Dependency Injection	A non-dynamic WAF allows injecting all kind of dependencies, which may be required by the application logic, into controllers.	p. 75	AVERAGE
C12	Method Signature Analysis	To relieve controllers of getting required objects themselves, e.g. by using dependency injection, or forcing them to implement specific interfaces, the WAF can analyze the method's signature to inject these objects into method parameters automatically.	p. 75	HIGH
C13	Abstraction over Servlet API	The WAF abstracts away the common Servlet API, simplifying the way requests can be parsed. This is usually the case for frameworks offering method signature analysis or related concepts.	p. 75	LOW
Input Validation				
C14	Manual Invocation	The validation of entities that get passed to the controller can be invoked manually, without calling the persistence context.	5.3.4	AVERAGE
C15	Command Objects	Validation for Command Objects without a database representation is available and can be invoked manually.	5.3.4	AVERAGE
C16	Black- and Whitelists	To avoid vulnerabilities in this regard, Command Objects bound to the controller should be secured of unwanted input using black- and whitelists.	5.3.4	AVERAGE

No.	Criterion	Description	Ref.	Influence
C17	Manual Retrieval of Results	The results of a manually invoked validation are available to the controller action, for the success or failure of the validation decides on subsequent operations.	5.3.4	AVERAGE
C18	Error propagation mechanism	In case of failing validations, the WAF makes the corresponding error messages available to the underlying view automatically. The developer only needs to forward the user to the correct page.	5.3.4	HIGH
C19	Internationalization	The messages returned to the user in case of failing validations are open for Internationalization.	5.3.4	HIGH
Context Scope Management				
C20	Conversation Scope	To keep data in scope over multiple screens, a special conversation scope is offered by the framework. Otherwise, the data would need to be stored in the session, which is difficult to clean up and can easily interfere with parallel processes by the same user.	5.3.5	HIGH
C21	Persistence Context management	To reduce the amount of database communications, and to avoid risky `merge()` operations, the WAF offers an easy way to set the persistence context to *extended mode*.	5.3.5	HIGH
C22	Flash scope	To make certain data survive a redirect or a new request, a special "flash scope" is offered by the WAF. This is very important for common use cases where the Redirect-after-Post pattern is used to display error or info messages in response to POST requests.	5.3.5	HIGH
C23	Accessing scopes	For being a recurring task, the access to the different scopes is simplified by the WAF, e.g. using method signature analysis or implicit objects.	5.3.5	AVERAGE
View Mapping				
C24	Convention over Configuration	To map views to controller actions, a default approach requiring no additional actions in the code is provided by the WAF.	5.3.6	LOW

No.	Criterion	Description	Ref.	Influence
C25	Return value approach	For being the simplest solution when conventions do not fit, the WAF allows defining the view to display using a controller method's return value.	5.3.6	AVERAGE
C26	Ease of Redirects	For being an important and recurring task, the invocation of redirects is eased by the WAF.	5.3.6	LOW

A.4 Development Support (D)

No.	Criterion	Description	Ref.	Influence
Project Generator				
D1	Contextual awareness	For the WAF's project generator keeps track of the current input session, it is able to offer special support regarding the current situation, e.g. by command or help suggestions.	5.4.1	LOW
D2	Round-tripping	The generator not only generates code for the developer, it is also able to react on the developer's actions outside the generator, to create additional code or remove old one.	5.4.1	AVERAGE
D3	Application awareness	The generator is designed in a way that is able to track the state of an application over multiple input sessions and can even react on changes that have been made to the application in the absence of a running generator instance.	5.4.1	AVERAGE
D4	Parallel usage in different projects	If the WAF is used for different web application projects in parallel, the generator remains usable for each of these projects throughout their development. The productivity gain of this feature obviously depends on the generator's overall functionality.	5.4.1	LOW-HIGH

No.	Criterion	Description	Ref.	Influence
D5	Input Comfort	The generator addresses usability by providing features like convention over configuration, tab completion, a help system etc. This is important to foster its usage, for a developer will then likelier use it again.	5.4.1	LOW
D6	Command Recording	The WAF allows recording sequences of commands issued to the project generator and replay them later in different projects or on different machines.	5.4.1	LOW
IDE Support				
D7	Compilation support	Web applications have a special structure an IDE needs to understand. The WAF provides a mechanism to assure this. This also includes a way to compile the application from inside the IDE. Sometimes, this requires additional IDE-capabilities (e.g., special plug-ins).	5.4.2	HIGH
D8	IDE launch commands	The WAF provides enough configurations so that application deployment becomes possible from inside the IDE.	5.4.2	LOW
D9	Plug-in: Project creation	A plug-in to create new web applications from inside the IDE exists. Depending on the WAF, this might even be an out-of-the-box feature of the respective IDE. feature may even come	5.4.2	LOW
D10	Plug-in: Project Generator replacement	A special plug-in replacing the common project generator with a new console inside the IDE is provided. This way, the WAF becomes completely manageable from inside the IDE.	5.4.2	LOW
D11	Plug-in: Artifact creation	A plug-in is provided through which particular WAF-specific artifacts can be created from inside the IDE. This feature then replaces certain commands provided by the project generator which no longer have to be remembered and typed.	5.4.2	LOW
D12	Plug-in: Artifact view	A plug-in that provides an optimized overview on the project by taking the application's artifacts into special consideration.	5.4.2	AVERAGE

No.	Criterion	Description	Ref.	Influence
D13	Plug-in: Advanced Error Detection	Artifacts, especially configuration and markup files, can be validated using special IDE plug-ins. This validation mechanism also addresses references to other components or artifacts defined inside these files.	5.4.2	AVERAGE
D14	Plug-in: Visual Editors	The creation of certain artifacts is simplified by providing special visual editors.	5.4.2	AVERAGE
Dynamic Language Support				
D15	Support of dynamic languages	The WAF supports the usage of dynamic languages, for many application logic tasks can be fulfilled much easier and faster leveraging features provided by dynamic languages.	5.4.3	HIGH
D16	DSL support	The WAF supports special Domain Specific Languages to be used with dynamic languages. This especially makes sense regarding WAF-specific artifacts like controllers or persistent entities.	5.4.3	AVERAGE

A.5 Build & Integration Support (I)

No.	Criterion	Description	Ref.	Influence
Build Management				
I1	Construction of directory layout	The WAF automatically generates all files and folders required to successfully deploy the application.	5.5.1	LOW
I2	Compile, Package	The WAF provides ways to compile and package the application. To this, special build scripts can be generated or the project generator may offer special commands.	5.5.1	HIGH
I3	Deploy	The deployment of a compiled package is possible using a special build script or project generator command.	5.5.1	HIGH

No.	Criterion	Description	Ref.	Influence
Environment Configuration				
I4	Support of environments	The WAF provides different environment modes, e.g., to test, develop or run the application in production. These environments are fully configured using meaningful default calibrations regarding logging, UI-stacktraces, database access etc.	5.5.2	HIGH
I5	Per environment configuration	All relevant configurations are applied on a per-environment basis and can be adapted easily.	5.5.2	HIGH
I6	Switch between environments	The WAF provides a fast and easy way to switch between different environments, e.g., using the project generator.	5.5.2	LOW
Dependency Management				
I7	Integration of basic dependencies	Those dependencies typically required by almost every web application are integrated by default or automatically when they are used for the first time. Obviously, the latter is the better option for it makes sure no unnecessary libraries and configurations are applied.	5.5.2	HIGH
I8	Automatic dependency-management	New dependencies are resolved automatically, without requiring the developer to touch configuration.	5.5.3	HIGH
I9	Integration support	The WAF eases the integration of new APIs and frameworks into the web application (e.g., by providing a plug-in mechanism).	5.5.3	AVERAGE
Plug-ins / Add-ons				
I10	Plug-in mechanism	Plug-ins integrate new functionality into the application and do (most of) the required configuration automatically. An in-built plug-in mechanism therefore is a first step to significantly reduce integration efforts in web development.	5.5.4	HIGH
I11	Availability of Plug-ins	A high amount of available plug-ins increases the chances of finding an appropriate one for the functionality currently required.	5.5.4	HIGH

No.	Criterion	Description	Ref.	Influence
I12	Central Plug-in repository	To be able to find them, all available plug-ins are stored in a central repository that can be freely accessed by everyone. This repository might be a web application itself or a place accessible using the project generator.	5.5.4	AVERAGE
I13	Installation mechanism	Once an adequate plug-in is found, the WAF supports its easy installation in a standardized way.	5.5.4	LOW

A.6 Testing (T)

No.	Criterion	Description	Ref.	Influence
General Testing criteria				
T1	Generation of test stubs	The WAF generator provides commands to generate empty stubs for unit, integration and functional test classes. This also includes their proper configuration within the system.	5.6.2	AVERAGE
T2	Invocation of tests	Either the WAF's project generator or the generated build files provide commands to invoke test-runs for all available test types.	5.6.2	AVERAGE
T3	Reports	After testing, reports about the test results are automatically generated. The information includes overall results as well as detailed information about particular test cases.	5.6.2	HIGH
Hot Deployment				
T4	Static resources	Changes to static resources like stylesheets or graphics can be hot deployed and take effect immediately.	p. 88	AVERAGE
T5	Views	The WAF supports the hot deployment of the particular view technology in use. Changes to the respective view then take immediate effect on the server.	p. 88	AVERAGE

No.	Criterion	Description	Ref.	Influence
T6	Method Body changes	Source code changes restricted to method bodies can be hot deployed. Method signatures or other direct class members do not need to be modifiable though.	p. 88	AVERAGE
T7	Class Structure Changes	Besides allowing method body changes, the WAF also supports the hot deployment of classes which structures have changed, e.g. regarding method signatures or instance fields.	p. 88	AVERAGE
T8	Model Changes	Changes to persistent entities can be hot deployed and do not require a server restart.	p. 88	AVERAGE
T9	Automatic invocation	When a file that is allowed to be hot deployed changes, this process is automatically invoked. This functionality often is provided in conjunction with the use of an IDE.	p. 88	LOW
Debug page				
T10	Stacktraces	When doing manual testing, exceptions will occur. The corresponding stacktraces are easily accessible without opening the server log files, i.e., by means of special debug pages.	p. 89	LOW
T11	Scope variable contents	For stacktraces often do not provide enough information for debugging, debug pages provide additional information regarding the application state.	p. 89	AVERAGE
Unit testing				
T12	Support for Unit tests	The WAF integrates a solution for writing and invoking unit tests.	5.6.2	HIGH
T13	Support for Test Doubles	The definition of test doubles (likes Mocks and Stubs) is supported by the WAF.	5.6.2	HIGH
T14	Pre-configured test doubles	Some artifacts, like for example controllers, often depend on special objects (e.g., user sessions, request parameters). For all these objects, the WAF already provides ready-made test doubles.	5.6.2	HIGH
T15	Support for web-specific artifacts	Testing web-specific artifacts can be eased by providing special APIs for them.	5.6.2	AVERAGE

No.	Criterion	Description	Ref.	Influence
T16	Automatic generation	If testing has a high priority in WAF's spirit, unit test stubs might be created automatically whenever a new artifact (like controller, entity or service objects) is created.	5.6.2	LOW
Integration testing				
T17	Special environment	Integration tests need to run inside a container. To accelerate the tests, this environment is restricted (e.g., provide no Servlet container) or even able to run in embedded mode.	5.6.3	HIGH
T18	Support of URL mapping tests	Tests of URL mappings show whether requests are correctly bound to controller actions and their parameters.	5.6.3	AVERAGE
T19	Support of user interactions test	Tests of user interactions allow analyzing responses based on given inputs, in a view-independent manner.	5.6.3	AVERAGE
Functional testing				
T20	Support of HTTP-based tests	HTTP testing approaches allow the easy generation of HTTP requests and provide ways to verify the response.	5.6.4	AVERAGE
T21	Support of UI Tests	UI-driven testing allows the easy creation of functional tests by means of a point-and-click interface.	5.6.4	AVERAGE