Framework Software Quality Analysis: A Case Study

Analyzing the software quality supported by a J2EE meta-framework

Tim Prijn, 15th of August
Master Course Software Engineering

Thesis Supervisor: Dr. J. Vinju

Internship Supervisor: Ing. B. Rodenburg

Company: Info Support

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Acknowledgements

This thesis is the result of my graduation project for the title of Master of Science at the University of Amsterdam.

This graduation project would not have been successful without the support of a number of people. At first, I would like to thank Jurgen Vinju for always asking the right questions and for his criticism on this thesis. Furthermore, thanks goes out to Bastiaan Rodenburg for helping and supporting me in my search for a research subject at the initial start of this project and for his comments on earlier drafts. Next on my list of people I would like to thank is Paul Bakker, a fellow student and colleague. Paul was at all times prepared to share his experience in developing J2EE based applications. He together with Jacco van Willigen also gave me valuable feedback onto a first version of this thesis. In addition, I would like to thank Info Support and a number of colleagues, which I will not all mention here. Rest assured, your efforts are much appreciated.

At last, I want to thank my parents, Kees and Willy and my girlfriend Maudie, for their continuous love and support throughout the past year.

Abstract

Lately more and more open source frameworks emerge that provide an alternative to the traditional J2EE technologies. Systems built on top of these frameworks have an enormous impact on non-functional requirements as scalability, performance, modifiability, availability etc.

Info Support is an IT service provider in the area of training, consultancy and management in the Benelux. One of the most important goals for Info Support is to be up-to-date with the latest technology in Java distributed development. Even more important is that they have knowledge of the software quality of applications developed with a certain framework. However, practice proved that it is not a straightforward task to evaluate and document the support of a framework for these qualities.

The objective of this thesis is therefore to define a method for analyzing the software quality of frameworks.

At first, we conducted a literature study to provide answers to a number of questions identified in cooperation with our key stakeholder. The primary contributions of this study are manifold. At first, we explained frameworks according to a meta-model, which shows an abstract overview of the common aspects between different frameworks. Secondly, we classified frameworks into three categories showing that J2EE is a middleware infrastructure framework. Then the different J2EE frameworks itself were categorized and we concluded this research is interested in J2EE meta-frameworks. These frameworks address development needs in all tiers of the J2EE reference architecture. Our literature study ended with an overview of software quality analysis methods, which provided the foundation for our approach.

With the knowledge gained by the literature study, we defined an approach named Framework Software Quality Analysis Method (FSQAM). FSQAM is a questioning technique in which scenarios are used to document the non-functional requirements. FSQAM consists of three phases. The first phase, "*collect concrete scenarios for the quality attributes"*, results in a list of scenarios that describe the non-functional requirements for J2EE applications. The second phase, specify a framework-based application, results in the implementation of an application built on top of the framework onto which the scenarios can be analyzed. Subsequently, during the third phase the scenarios are actually analyzed against the framework-based application. For each scenario is explained how developers need to configure the framework to use its functionality, followed by an overview of how this affects the software quality.

The next phase of our study was the application of FSQAM to the Spring J2EE meta-framework. The results of this phase were both of practical and scientific use. With respect to the former, analysis of the scenarios selected by our key stakeholder pointed out that applications built on top of Spring are highly modifiable, testable and usable. With respect to the latter we also noted that most of the framework participants, that enhance usability, could proof a risk for modifiability when applied excessive. In addition we recognized that the increase in modifiability, testability and usability is mostly achieved at the cost of performance. However, future work should include measurements that exactly determine this impact.

With regard to the scientific contributions of applying FSQAM we identified a number of aspects to consider. At first, before the collected scenarios are analyzed they are prioritized by the stakeholders. This ensures that the precious analysis time is spent best. However, choosing different scenarios could lead to different results. A second observation was that the implementation of a framework could affect how usable it is for developers and should therefore be addressed. This implies that one has to develop an application on top of the framework, thereby also providing a source of reference for others.

In conclusion we believe that FSQAM is successfully applied to the case study. The stakeholders, specially the key stakeholder, has indicated he is satisfied with the results. That is; not only the results regarding the case study, but also the results with respect to the definition of an approach and the conducted literature study.

List of Contents

Chapter 1

Introduction and motivation

1.1 Context

In the 1980's, personal computers became gradually faster against lower costs, which enabled the extensive use of distributed computing. However, computer vendors continued producing hardware, operating systems and network protocols that could not interoperate with that of their competitors. For developing organizations this differentiation in platforms presented problems regarding distributed computing. They often needed to invest in several computing platforms to accomplish their goals.

Sun Microsystems addressed distributed communication between different platforms in the Java programming language. Once a Java application is developed, its code can be used on any hardware platform for which a Java Virtual Machine (JVM) exists. Additionally, remote communication between Java applications occur through a communication protocol named Remote Method Invocation (RMI), thereby allowing applications executed on different platforms to communicate.

By 1998, Java gained so much support of developing organizations and the open source community that its set of features and possibilities became too large and complex. Sun then decided to repackage the Java specification into three types: Java 2 Micro Edition (J2ME), Java 2 Standard Edition (J2SE) and Java 2 Enterprise Edition (J2EE). The latter being the focus of this research work.

J2EE provides an industry-standard specification for the software community to more easily build distributed object systems using the Java programming language. J2EE soon gathered popularity in Internet-time when all kinds of e-commerce applications needed to be developed. These systems are mostly developed to attract large volume of hits. There are, for example, sites that receive millions or many millions of requests a day, from which a lot arrive at the same time during peak hours. Among examples of failing stories is the Wimbledon tennis tournament site that experienced 7000 hits per second [BCK03]. This kind of software has an enormous impact on non-functional requirements as scalability, performance, availability and security. Sun Microsystems, in developing J2EE frameworks, aimed to provide a basis for technology that supports the construction of such systems. This basis consists of several frameworks, of which each addresses specific needs as web development, persistency etc., that together make up the J2EE specification.

Several years later a new trend is released: developing open source alternatives to the mainstream J2EE technologies. This is motivated by the heavyweight complexity of traditional J2EE frameworks. The trend towards developing open source frameworks began in 1999-2000 and filled needs for developing the web layer. Subsequently, all kinds of open-source framework substitutes were created for the traditional J2EE persistency frameworks. Another sub part of the J2EE specification was now within the domain of open source frameworks. However, large gaps remained as for instance with respect to implementing business logic. Currently, more open source frameworks arise that provide an alternative for the entire J2EE specification. Examples include, Spring [JHA05] and Pico[1].

Spring and Pico are application frameworks that aim to help structure whole applications. According to [Mil06] they are also called J2EE meta-frameworks, because of their strong reliance on integration with other frameworks.

As mentioned, support for the non-functional requirements is an important aspect in developing distributed (web) applications. Info Support is an IT service provider in the area of training, consultancy and management in the Benelux. For Info Support one of the most important goals is to be up-to-date with the latest technology in Java distributed development. Even more important is that they have knowledge of the

[1] http://www.picocontainer.org/

software quality of applications developed with a certain framework. However, practice proved it is not a straightforward task to analyze and document the support for qualities of a framework.

All together the objective of this thesis is to define a method for analyzing the software quality of J2EE meta-frameworks.

1.2 Scope

Info Support needs a method for analyzing the software quality of J2EE meta-frameworks. The intention of the method should not be to evaluate frameworks against a set of requirements. Instead, the analysis should show how the implementation of the framework and the manner in which it provides its functionality affects the software quality of applications developed on top of it.

1.3 Research question

Info Support needs a method for analyzing and reporting the software quality of J2EE meta-frameworks. This method should address the need to enlarge customer's confidence in recommendations provided by Info Support with regard to the choice of framework. The research question that follows from the previous is:

"*How to analyze the software quality of a J2EE meta-framework?*"

The question consists of two parts: at first a structured analysis method has to be determined to gain insight into the software qualities supported by the framework. Secondly, a way to document the outcome of the analysis phase has to be specified.

The software quality in this thesis is defined as in Dobrica *et al.* [DN02]: the software quality of a system represents the degree to which software possesses a desired combination of quality attributes. The quality attributes of importance are part of the analysis method and how to determine and document them is discussed in Chapter 5.

To be able to answer this research question we identified a number of sub-questions that by answering them should provide us with the knowledge necessary to define an approach. The questions are ordered per topic:

- Frameworks
  - Q1. How can a framework be defined?
  - Q2. Which parts define a framework and how can these be adapted to the developers needs?
  - Q3. Which benefits are gained when using a framework?
  - Q4. Are there common criteria a framework should meet in order to be successful?
  - Q5. There are lots of different types of frameworks that are classified by literature in different ways. Can frameworks be classified and in which category can J2EE be positioned?
  - Q6. Does literature also provide requirements for the category J2EE frameworks belong too?
- J2EE
  - Q7. What is the reference architecture for J2EE applications?
  - Q8. Can J2EE frameworks itself be classified?
  - Q9. What are the high-level goals set out by the J2EE specification?
- Software quality analysis methods
  - Q10. How to define software quality?
  - Q11. How to document non-functional requirements to evaluate the software quality?
  - Q12. Is software architecture know-how applicable for determining the software quality of frameworks?
  - Q13. Are there existing software quality analysis methods that can be applied to frameworks?

1.4 Research approach

Figure 1 shows how the current research is conducted. First, a literature study is performed into the topics: Frameworks, J2EE and Software Quality Analysis. The results of this study are published in respectively Chapter 2 (Q1 – Q6), Chapter 3 (Q7– Q9) and Chapter 4 (Q10–Q13).

This literature study is useful in more ways. First, this study provides the foundation of the current research in that the proposed method is partly based on this study and for the rest on own insights. Secondly, this study provides the background and terminology used in the remaining chapters.

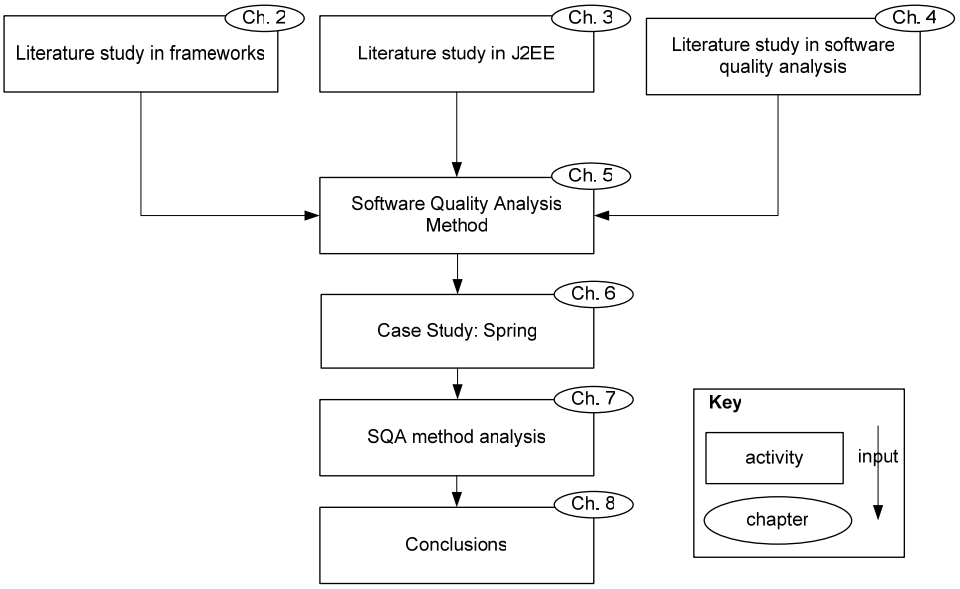

Figure 1: Research approach

The proposed method, which is based on own insights and the literature study, is subsequently presented in Chapter 5. Thereafter, Chapter 6 discusses the application and results of each phase of the analysis method to the Spring framework. In addition, Chapter 7 provides a discussion of the findings with respect to the usage and results of the proposed analysis method. Chapter 8 concludes this thesis with an overview of the primary contributions and provides direction for future work.

1.5 Contribution

The scientific contribution of this research work is the definition of a method for determining the software quality of J2EE meta-frameworks. During the literature study, the present author found little to no methods that described such a method and its applicability in a case study. The contribution from a practical point of view is the application of the defined approach to the Spring J2EE framework, of which the outcome is also twofold: at first the results of the analysis can be used to validate the defined approach. Additionally, the outcome is used by Info Support to provide their customers with a recommendation regarding when the use of the Spring framework is justified.

Chapter 2

Frameworks

This chapter discusses frameworks in general. It serves as the foundation on which the current research is based and it will identify general criteria and benefits of using frameworks. At first, a definition for frameworks is provided followed by a discussion of what each framework is made up of. The third paragraph then shows the benefits and issues of using frameworks. Subsequently, we present a framework classification and show the category J2EE frameworks belong to. The last paragraph shortly summarizes the lessons learned in this chapter with regard to the sub questions of this topic.

2.1 Framework definition

In literature many definitions of a framework exist. In [Hau97] the author defines a framework as: "*a reusable design expressed as a set of abstract classes and the way their instances collaborate.*" The author does not mention a framework should provide functionality for a common purpose. Larman [Lar02] defines frameworks as: "*a framework is an extendable set of objects for related functions*". This definition takes in account that a framework should provide developers with functionality for a general goal, it however does not mention frameworks are a reusable design.

In [MFB02] the authors use the same definition for a framework and also define the related set of functions as: *"a set of related functions is the set that defines the area of expertise or competencies of the framework*". This set is also called the domain. [MFB02] furthermore argues that frameworks also include a computational architecture of the reference architecture.

A reference architecture is the mapping of a reference model to architectural elements and the data flows between them. A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem. Reference models are for example, the standard parts of a compiler and how they interact [BCK03]. These reference models are a description of the domain it represents. Reference architectures in the sense of Bass *et al.* [BCK03] only constitute of the architectural blueprint. With frameworks we are also interested in the implementation, the computational architecture of this reference architecture. In other words, the framework should be a piece of software that helps us accomplish certain tasks.

The above definitions were the basis for the one used in the present research: "*A framework is a reusable, extendable set of objects for related functions that characterizes the area of expertise along with a computational architecture of its reference architecture*."

2.2 Framework metamodel

Frameworks come in many different forms. Mili *et al.* [MFB02] observed that frameworks generally consist of the facets shown in *Figure 2*. The participants in a framework fulfil a particular role. In J2EE terms an example of such a role could be the data access technology. These roles often share a relation with another role. One could imagine that a transaction model is used, together with the data access component to accomplish writing data in an atomic fashion. A role in a framework is described in terms of an interface that the participant must support. The interface consists of a set of attributes and method signatures which participants fulfilling that role must implement. Data access methods for example are mostly described by methods that are able to write or read data to and from the database.

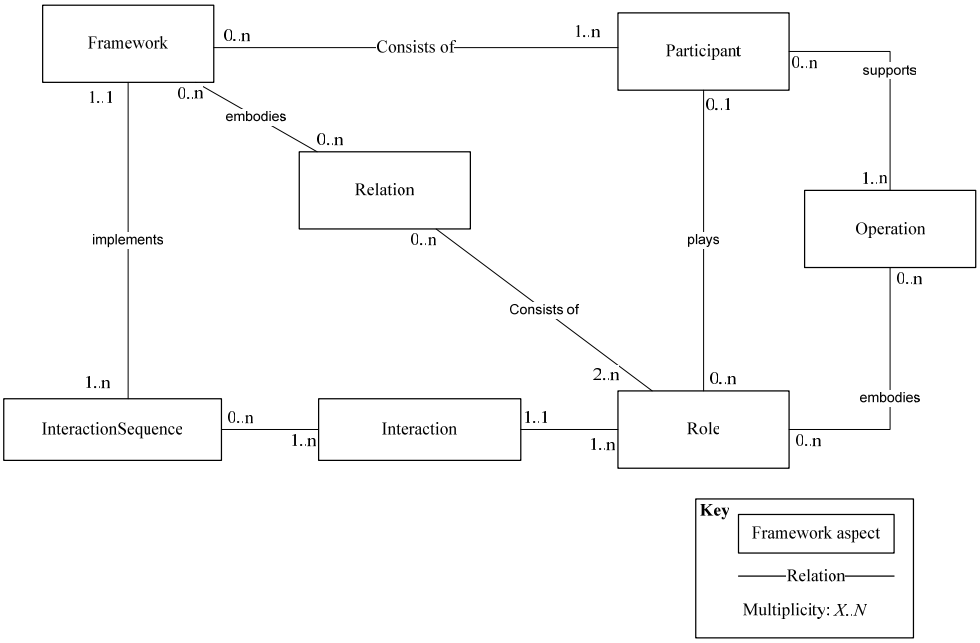

Figure 2: Framework metamodel [MFB02]

An interaction sequence represents communication between one or more roles. This could for example be the interaction between the transaction participant and the data access participant. An interaction sequence consists of one or more interactions. The results of an interaction could namely be the input for another interaction.

Dependent on the size and requirements of a framework the metamodel participants are different. In case of typical small scale applications the participants can be, dependent on the language used, classes or modules. Operations then often consist of methods or functions. The interaction mechanisms are often close to the language, such as method or function calls. In some cases the framework needs to support distributed services and then remote procedure calls, implemented by frameworks such as RMI or CORBA (see §3.1), are used. The traditional J2EE provides distributed services through RMI.

In case of large frameworks, participants can be sub-frameworks. In the traditional J2EE technologies frameworks often only address the needs for a particular tier in a three tier reference architecture (see §3.1). For J2EE among the following sub frameworks are identified: Servlets, Enterprise Java Beans (EJB) and Java Database Connectivity (JDBC) [Joh02]. Each framework addresses specific needs for a tier. Operations in sub-frameworks are represented by a set of interfaces. Another common case for large scaled frameworks is that they need to interact with applications written in a different language. Therefore language independent interaction mechanisms are used, such as CORBA or web services [J2EE03].

The methods defined by the user to tailor the participants and interaction sequences according to the needs of the framework user are often called from within the framework itself. This concept is termed the Hollywood principle: "*Don't call us, we'll call you*". This means that the user-defined classes (for example, new subclasses) will receive messages from the pre-defined framework classes [Lar02]. This inversion of control gives frameworks the power to serve as extensible skeletons. The extension points of a framework, the points where user defined code can be added are called hooks or also often named hotspots [Hau97]. Hooks can be as simple as inheriting from a framework class, or as complex as adapting an interaction sequence according to a configuration file. How to use these hooks depends on the type of framework: Blackbox or Whitebox. The difference in these framework stems from two different ways of object-oriented thinking.

Gamma *et al.* [GHJV94] address this issue. They argue that reuse by sub classing means that the internals of the parent are often visible to the sub class. Object composition is the Blackbox approach, because the internals of classes are not visible to each other. Here new functionality is obtained by letting classes communicate through well defined interfaces.

In [GHJV94] the authors assert that one should favour object composition over inheritance. The inheritance approach comes with a couple of drawbacks. At first it is not possible to change the implementations inherited from parent classes at run-time. Secondly, inheritance often exposes a subclass to details of its

parent; one can argue it breaks encapsulation. Thirdly, reusing subclasses is difficult when one aspect of the inherited implementation is not appropriate for the new context. At last, multiple inheritance is not possible in some languages. This means that developers may need to adapt their domain design according to the limitations set out by the framework. Object composition is defined at run-time through interfaces which makes it possible to replace any object by another as long as it has the same type. In an ideal situation a new component should not be developed but assembled by object composition.

The discussion Whitebox versus Blackbox frameworks is similar[JF88]. Ideally, one would like to know as less as possible of the internals of a framework and use the framework through well defined interfaces. If an application requires the creation of many subclasses of framework classes, it can be difficult for a new programmer to learn the design of an application well enough to change it [McC04]. However, because reuse of Blackbox frameworks is limited to the functionality already implemented in the framework, this practice can seldom be applied in a pure fashion. A lot of mixtures exist in which inheritance is used with care.

2.3 Framework benefits and issues

In literature numerous benefits can be found [Joh05; HF98; Hau97, FS97]:

- *Increase in developer's productivity*.
- *Reuse of (tested) code*. The framework provides functionality that can be reused by the developer. Open-source based frameworks are often better tested, because it is used by many developers under different circumstances.
- *Simplifies development* because developers do not need to work directly with low level APIs.
- *Developers can focus on their primary task* because the framework offers a general set of related functions. For instance, when developing an enterprise java bean (see §3.1) developers can concentrate on implementing the business functionality without having to be occupied with technical concerns as pooling and the life-cycle of the bean.
- *Promote best practice* through examples and documentation.
- *Increase software quality.* Frameworks encompass architectural design decisions, which are often based on architectural patterns and design patterns (best practices). In general, these kind of patterns are best practices of which one knows the qualities it addresses.

The most found and widely advocated benefit of framework usage is the increase in the developers productivity. This is primarily accomplished by reuse of (better tested) code and because developers can better concentrate on their primary tasks.

Before a framework accomplishes its benefits, a couple of issues should be overcome. The study in literature pointed out the following high-level goals for a framework to be successful [MBF99; MFB02; Lar02; FS97]:

- *Frameworks have to be suited for integration with other frameworks*. The major problem with integration is determining the control flow, considering each framework uses its own inversion of control mechanism.
- *Frameworks have to be maintainable*. When using an externally developed framework one must rely on their maintenance activities. Adapting an open source framework is an option, but developers should keep in mind that updating to a new release of the framework should still be an option.
- *Frameworks have to achieve its primary purpose*. Important for frameworks is that they focus on their main task and that the core is stable. This is funded on a citation of Larman [Lar02]: *"A framework should provide classes and interfaces that collaborate to provide services for the core, unvarying parts of the framework"*. The hot spots, the variability points, should provide enough flexibility to accomplish the required domain functionality. In practice it is hard to predict what should be in a framework and what should not be.
- *Frameworks have to be portable*. Platform independence is many distributed application environment an important aspect, because it enhances the ability to use software systems in different hardware environments.
- *Frameworks have to minimize the learning curve*. Documentation and support is vital to the success of a framework. In general learning to use frameworks is a considerable investment. Documentation in the form of a manual, tutorials and news groups can significantly help in learning to use the framework.
- *Frameworks have to minimize the effect on debugging and testing.* Frameworks are harder to debug and test. The first reason is that the framework usually abstracts away from application-specific details in a white- or black box manner. This mostly leads to the case that the users' code cannot be tested without the use of the frameworks' code. It is also hard to distinguish bugs in the framework from bugs of the users' code. Another cause that makes debugging and

testing hard is that one does not have explicit control over the control flow due to the inversion of control mechanism. This mostly increases the difficulty of stepping through a program in debug mode since the control flow of the application is driven by call-backs to the framework and developers may not understand or have access to the framework code.

- *Frameworks preferably provide infrastructure not generation*: a framework provides the plumbing, it does not generate code. If it does generate code one can expect to have problems during maintenance.

2.4 Framework classification

Frameworks can be classified by domain or scope in the following ways [FS97]:

- *Domain* is subdivided in the business and application domain. The business domain is problem space related; a banking application framework, for example, models the domain classes, their interrelationships and their interactions. The application domain is more design oriented and addresses architectural issues. Well-known examples include Model-View-Controller and RMI (see §3.1).
- *Scope* is classified in System Infrastructure frameworks, Middleware Integration frameworks and Enterprise Application frameworks. System Infrastructure frameworks simplify the development of portable and efficient system infrastructure such as operating system-, communication-, user interfaces- and language processing frameworks. These kinds of frameworks are mostly used within an organization. The second type, Middleware integration frameworks, is commonly used to integrate distributed applications and components. Examples include, CORBA, COM+ and EJB [Zar04]. Finally, Enterprise application frameworks address broad application domains (such as telecommunications, avionics, manufacturing and financial engineering). Enterprise application frameworks are often composed of both business and application domain functionality.

Figure 3 represents how J2EE is positioned. Considering that J2EE frameworks are developed to be used in different domains, such as banking, retail and financial, it cannot provide business domain specific functionality. J2EE applications are used for building distributed applications. Furthermore, the motivation for this framework stems from the fact that building distributed systems should be possible for every software development organization and it could therefore not belong to the system infrastructure framework. Zarras [Zar04] also classifies J2EE as a middleware infrastructure.

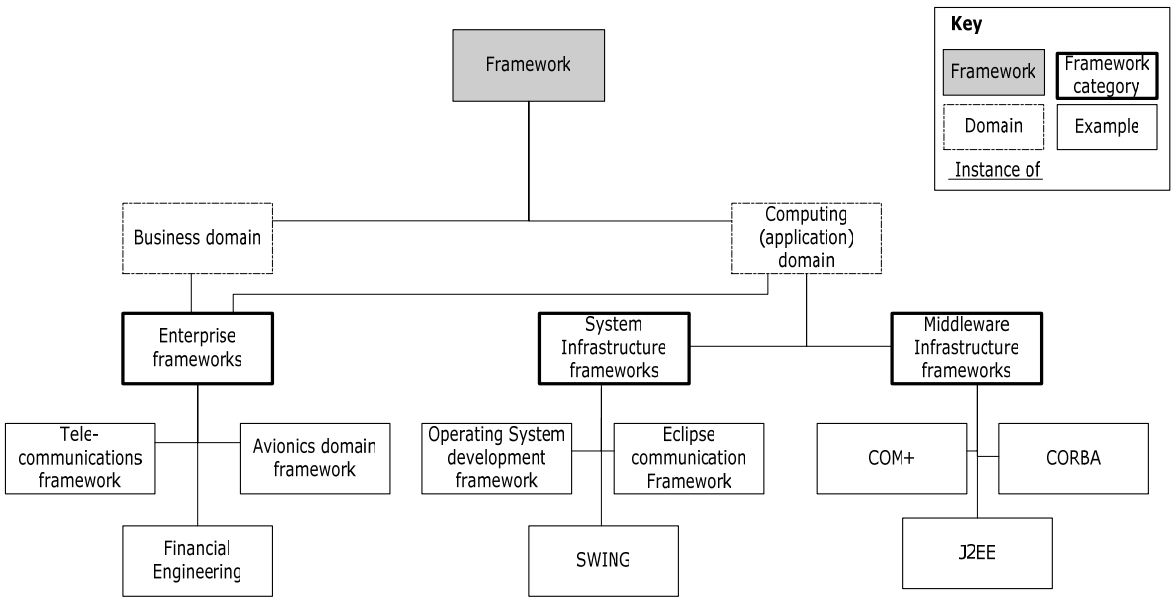

Figure 3: J2EE positioned in the application framework context

As discussed in §1.1, J2EE frameworks itself can be classified. However, this discussion is postponed to §3.1.

2.5 Requirements for middleware infrastructure frameworks

The previous section concluded that J2EE is categorized as a middleware infrastructure framework. Study in literature resulted in a paper that discussed a method for evaluating middleware infrastructures, according to a couple of key-requirements. Zarras [Zar04] proposed a comparison framework that consists of a set of key requirements typically imposed by distributed applications over the middleware. Their comparison framework consists of a generic architectural style that encompasses the key requirements and an identification of fundamental services that the infrastructure should provide.

In [Zar04] the author identified the following set of key requirements:

- Openness: The middleware infrastructure should enable extending the applications built on top of it in various ways. (adding, removing, upgrading, composing services etc.)
- Scalability: The middleware infrastructure should facilitate the effective operation of the application at many different scales.
- Performance: The middleware infrastructure should enable the efficient and predictable, if needed, execution of the applications that are built on top of it.
- Distribution transparency: is the property that determines if the application perceived by users or developers as a whole rather than as a collection of independent constituent element. This requirement can be divided into a more specific set:
    - Access transparency: The infrastructure should enable accessing local and remote application elements in the same way.
    - Location transparency: the infrastructure should enable accessing the application elements without knowledge of their physical location.
    - Concurrency transparency: the infrastructure should allow concurrent processing on resources without interference.
    - Failure transparency: the infrastructure should enable service provisioning despite the occurrence of failures.
    - Migration transparency: the infrastructure should provide means for changing the location of elements of the application without compromising the application's correct operation, i.e. without affecting the elements that depend on the migrated elements.
    - Persistence transparency: the infrastructure should provide means for coordinating the execution of atomic and isolated transactions.

These key requirements will not suffice for the current research. The problem is that the focus of Zarras is on the non-functional requirements only with regard to the communication between participants in a framework and the communication between different applications supported by the framework. J2EE-meta frameworks often provide a greater amount of functionality then only with respect to communication. The information in Zarras is however valuable, because it provides partly what we want to gain insight into in this research.

2.6 Summary

This chapter is the result of the literature study phase and provided a basic knowledge of frameworks in general. Additionally, it sets out the terminology used in the remaining chapters. Frameworks are defined (Q1). Subsequently, a basic abstract understanding is provided in which participants, roles and interaction sequences are seen as the basic building blocks on which a framework relies. Common characteristics among frameworks in general were recognized to consist of: inversion of control, hot spots and the black- or white box approach in which to configure or use the participants (Q2).

Thereafter, a list of benefits found in literature was provided followed by the statement that the increase in developer's productivity can be seen as the most important benefit (Q3). This chapter was then continued with a list of issues with framework development that need to be dealt with in order for the framework to become successful in its field. Most notable was the effect of inversion of control on testing and debugging the application (Q4).

Then we questioned if frameworks can be subdivided according to some criteria (Q5). Literature provided two ways in which this can be achieved: by domain and by type. We concluded that J2EE frameworks only address the computing domain and within this domain, J2EE is seen as a middleware infrastructure. In addition a number of requirements for these type of frameworks were listed (Q6). This list can be used in the case study to determine the high-level goals of the framework that needs to be analyzed. The following chapter will continue with a discussion of J2EE and then some key requirements for middleware infrastructure frameworks are identified that can be used in this research.

Chapter 3

J2EE

This chapter provides the introduction and terminology of the J2EE technology. At first, we provide a summary of the reference architecture propagated by J2EE. In addition we list the high-level goals of J2EE identified during the literature study. As concluded in Chapter 2, J2EE can be classified as a middleware infrastructure and thus we also list the requirements found for these kinds of frameworks. In summary this chapter provides answers to questions Q7 up till Q9.

3.1 Introduction in J2EE

As described in §1.1, with the advance of the Internet and especially the World Wide Web in the early 1990s the need to define standard methodologies to design and build efficient, robust and maintainable (web) applications has become increasingly important. J2EE offers a technical platform and a set of frameworks to design these kinds of applications that require high availability, manageability, security, scalability and short time-to-market.

To accomplish this goal J2EE is made up of numerous components and services. The following paragraphs show a part of the reference architecture (see §2.1) of J2EE applications starting with Figure 4. This figure provides an architectural view in which the emphasis is on containers. Note that this figure shows the logical relationships of the elements; it is not meant to imply a physical partitioning of the elements into separate machines, processes, address spaces or virtual machines [J2EE03]. The containers are also called J2EE runtime environments that provide services to the application components, the ones in the upper half of the rectangle. The services available are denoted as grey coloured rectangles.

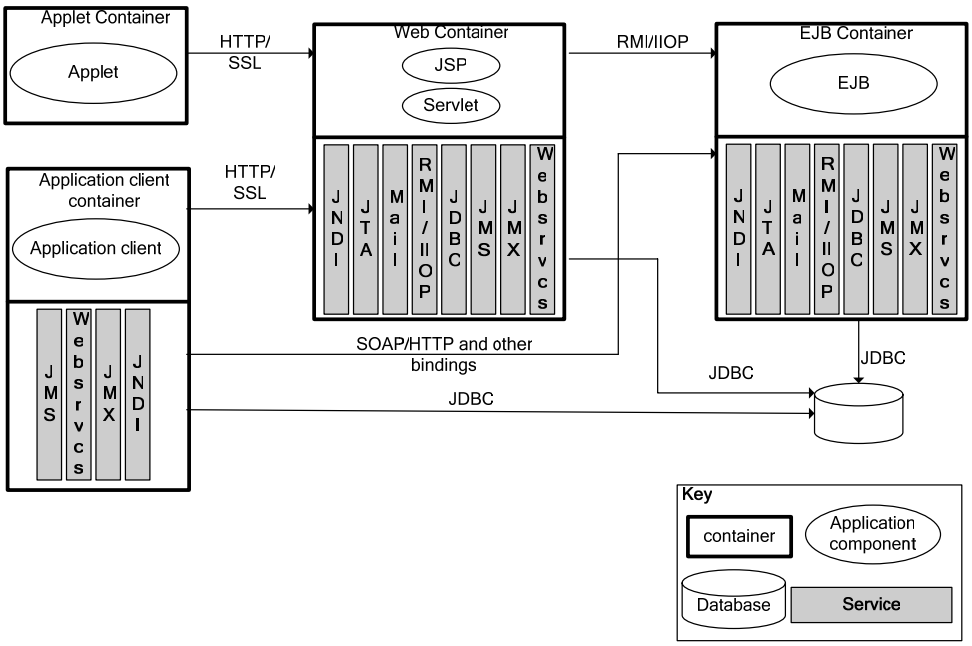

Figure 4: J2EE reference architecture; the container model [J2EE03]

It should be noted that Figure 4 is a partial representation of the J2EE architecture, only those were chosen that have relevance with further reading of this thesis. A short description of each sub-framework/service can be found in the First Addendum.

While the container model, Figure 4, focuses more on the runtime context, the tier model shown in Figure 5, describes the division into architectural layers [J2EE03]. J2EE uses a multitiered distributed application model. This means that J2EE applications are often divided in three tiers and each tier could execute on a separate hardware device. For now a simplified version of this architecture, presented in Figure 5, will be used.

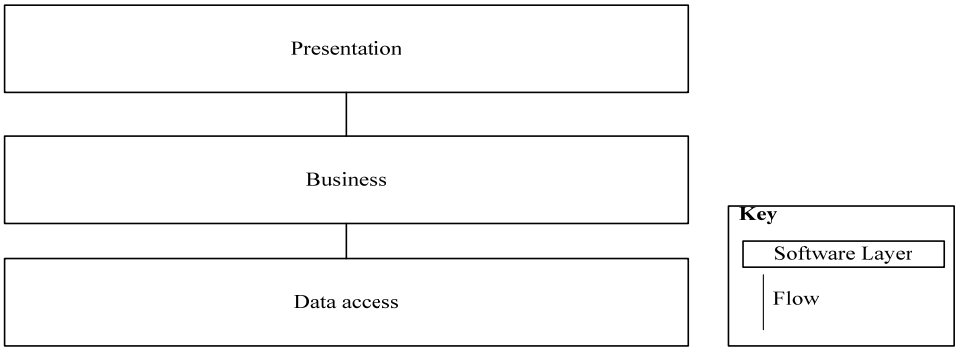

Figure 5: J2EE reference architecture; simplified development model

The presentation tier is used for representing content to the requestor and can be, but not necessarily is, web-based. If the presentation tier is web-based, it runs a web container to handle client requests and responds to these requests by invoking J2EE Servlets or JavaServer Pages. Obviously, the client uses a browser to send to and receive requests from the web container. The Servlets or JSP pages are the point of communication between the Java code and the HTML requests and responses. Other kinds of presentation tiers, as applets or Java applications, directly communicate with the EJB container, through for example web services, as shown by Figure 4. The open-source framework alternatives for this tier are web frameworks as for instance Struts[2] and JSF[3].

The second tier, the business, contains the business logic that solves or meets the needs of a particular business domain such as banking, retail or finance. The business logic is often realized in Enterprise JavaBeans. These beans are hosted by an EJB container that provides a number of services such as transactions, life-cycle management and security, as can be seen in Figure 4. During the literature study we did not find any specific open-source frameworks that only address issues in this tier. However, there are many alternatives that address specific issues also offered by J2EE as for example caching, transactions and pooling frameworks. On the other hand, there are the J2EE meta-frameworks (§1.1) that address all issues in J2EE development, thus also for implementing the business logic, such as Spring [JHA05].

Finally, the data access tier handles database communication or communication with enterprise information system software such as enterprise resource planning (ERP), mainframe transaction processing, database systems and other legacy information systems [Paw01]. For communication with a database server J2EE offers JDBC and Entity beans. The latter being a specific instance of EJB for having a bean synchronized with its persistence store. Many open source frameworks as an alternative for J2EE in this tier are available. Among the most well-known are Hibernate[4] and IBatis[5].

Regarding the classification of J2EE frameworks, we mostly speak about sub-frameworks that only address a single tier; web-only, persistency-only and remaining frameworks. The latter addresses more low-level issues as caching, transactions and pooling. Besides these single-tier only frameworks, another category consists of the frameworks that address all tiers and issues in J2EE development; the meta-frameworks (§1.1). However, often these frameworks rely on integration with sub-frameworks. For instance the Spring J2EE meta-framework can be configured to use Hibernate as its persistency framework. According to [MFB02] this is a normal phenomenon because the scale of frameworks often results in the fact that the participants in a framework are subsystems or other frameworks.

[2] http://struts.apache.org/
[3] http://java.sun.com/javaee/javaserverfaces/
[4] http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf
[5] http://ibatis.apache.org/

3.2 High-level goals addressed by J2EE

The high-level goals J2EE tries to meet are the following [BCK03; J2EE03]:

- *Make it easier to write distributed object-oriented applications.* Application developers do not have to deal with low-level details of transaction and state management, multi-threading, and resource pooling.
- *Scalable applications.* J2EE applications should be scalable without human intervention.
- *Provide authentication mechanism.* J2EE based applications should authenticate users and protect against unauthorized access to data.
- *J2EE application should be portable to other platforms.*
- *J2EE application should be manageable.* Application developers should be provided with facilities to manage common services such as transaction, name services, and security.
- *J2EE specification* should be detailed enough to provide meaningful standard for component developers, vendors, and integrators, but general enough to allow vendor-specific features and optimizations.
- *Separation of concerns.* Provide complete transparency of implementation details so that client programs can be independent of object implementation details (server-side location, operating system, vendor etc.).
- *J2EE remote communication interoperation.* J2EE should support interoperation of server-side components implemented on different vendor implementations; allow bridges for interoperability of the J2EE platform to other technologies such as CORBA and Microsoft component technology.

When analyzing the software quality of J2EE meta-frameworks these requirements should be addressed in order to determine the software quality.

3.3 Summary

This chapter provided the context and terminology of J2EE development that will be used throughout the remaining chapters. At first, an introduction to the traditional J2EE reference architecture was provided in which the use of components, containers and services was explained. Subsequently, we showed that most J2EE applications are based on three tiers and shortly discussed the purpose of that tier (Q7). On top of the standard J2EE technologies we showed some of the open source alternatives available, concluding with a classification of J2EE frameworks into: single-tier (web-only, persistency-only and remaining frameworks) and the meta-frameworks that address all tiers and issues in J2EE development (Q8). Additionally, a list of high-level goals of J2EE was provided that can be used as input for determining the software quality of a J2EE meta-framework (Q9).

Chapter 4

Software quality analysis

This chapter provides the results of a study in literature to software quality analysis. First, a definition for software quality analysis is provided. Then a discussion on which level software quality can be analyzed. Then several evaluation methods and categories are discussed. The results of this chapter are the basis of the proposed method.

4.1 Software quality

As described in §1.3, the software quality of a software system is defined as the degree to which software possesses a desired combination of quality attributes [DN02]. In literature many definitions for a quality attribute exist. Zhu *et al.* [ZBJ04] defines a quality attribute as "*A quality attribute is a non-functional characteristic of a component or a system*". This definition is only applicable for software and not for related work products, as in documentation etc., which is an important criteria of a good framework (see §2.3). It furthermore does not specify for whom its quality is important and remains vague. Therefore, a quality attribute in the current research is defined as in [BCK03]: "A quality attribute is a *property of a work product or goods by which its quality will be judged by some stakeholder or stakeholders*".

Literature research identified three issues with quality attributes:

- Different definitions of quality attributes exist [BCK03]. For example, maintainability, flexibility and modifiability are defined as follows:
  - Maintainability is a set of attributes which have a bearing on the effort needed to make specified modifications [DN02].
  - Modifiability is the ability to make changes quickly and cost effectively [BCK03].
  - Flexibility is the ease with which a system or components can be modified for use in applications or environments other than those for which it was specifically designed [IEEE90].
- Each attribute community developed its own vocabulary. The performance community has "*events*" arriving at the system, the security community has "*attacks*" arriving at a system etc. All of these "*things*" may refer to the same occurrence, but use different terminology.
- Quality attributes have overlapping definitions [BCK03]. Attributes are not discrete or isolated. For example, availability is an attribute in its own right. However, it is also a subset of security (because denial of service attack could limit availability) and usability (because users require maximum uptime). It is hard to determine to which quality a certain aspect belongs.

The first two problems can be addressed by quality attribute scenarios. The last problem is solved by shortly discussing the underlying concerns of the attribute. A quality attribute scenario is a quality-attribute-specific (non functional) requirement, that consists of the following six parts [BCK03]:

- Source of stimulus: some entity (a human, a computer system, or any actuator) that generated the stimulus.
- Stimulus: the stimulus is a condition that needs to be considered when it arrives at a system.
- Environment: the stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- Artefact: some artefact is stimulated. This may be the whole system or some pieces of it.
- Response: The response is the activity undertaken after the arrival of the stimulus.
- Response measure; when the response occurs, it should be measurable in some fashion so that the requirement can be tested.

In [BCK03] two kinds of scenarios are presented: general and concrete scenarios. The general scenarios are system independent. The concrete scenarios are specific for certain software architectures. Bass *et al.* [BCK03] identified for each quality attribute a table that gives possible system-independent values for each of the six parts of a quality scenario. Table 1, for example, lists the possible input for each component in a modifiability general scenario.

From this table one could identify for example the following concrete scenario: "*A developer wishes to change the user interface to make a screen's background colour blue. This change will be made to the code at design time. It will take less than three hours to make and test the change and no side effect changes will occur in the behaviour.*" [BCK03]

Table 1: Possible input modifiability general scenario [BCK03]	
Component	Possible values
Source	End user, developer, system administrator
Stimulus	Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artefact	System user interface, platform, environment, system that interoperates with target system
Environment	At runtime, compile time, build time, design time
Response	Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
Response Measure	Cost in term of number of elements affected, effort, money; extent to which this affects other functions or quality attributes

These table generation scenarios can be used for generating a large number of system-independent, quality attribute specific scenarios. A subsequent step would make them system-specific. This also implies that one general scenario could have more system-specific scenarios. The authors of [BCK03] also identified system independent solutions to implement certain scenarios. These solutions are called tactics.

A tactic is defined as: "*a design decision that influences the control of a quality attribute response.*" These tactics can be used in the proposed method, because the implementation of framework participants will also encompass tactics.

During recent years the notion of software architecture has emerged as the appropriate level for dealing with software quality. That is, in large systems, the achievement of qualities such as performance, availability, and modifiability depends more on the overall software architecture than on code-level practices such as detailed design, algorithms, data structures and so forth [KKB98].

4.2 Software architecture

A number of different definitions exist for software architecture. A well-known definition is the one defined by the Software Engineering Institute [BCK03]: *"The software architecture of a program or computer system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*"

As stated by Vliet *et al.* [LRV01], this definition does not take in account that systems communicate with external systems, which often is the case in business systems. The latter type of systems is also often the type of systems build with J2EE frameworks. Therefore the definition does not suffice for our practice and the one provided by IEEE is used: "*Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution*."

The software architecture of a system is important for three reasons [BCK03]:

- *Communication among stakeholders.* Software architecture represents a common high-level abstraction of a system that most if not all of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.
- *Early design decisions.* The software architecture of a system is the earliest artefact that enables the priorities among competing concerns to be analyzed, and it is the artefact that manifests the concerns as system qualities. The trade-offs between performance and security, between maintainability and reliability, and between the cost of the current development effort and the cost of future developments are all manifested in the architecture.
- *Transferable abstraction of a system*. Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements and can promote large-scale reuse.

On top of the motivation already presented, software architecture is also important for defining a framework software quality analysis method, because of:

- *J2EE meta-frameworks encompass architecture* and address all three tiers, communication between these tiers and communication with external systems, all being architectural issues.

- *Little to no existing framework software quality analysis methods*.
- *Reuse of software architecture evaluation know-how*. Much research has been done and a lot of experience has been gathered and published in the field of software architecture evaluation, some of which can be reused.
- *Frameworks and software product lines overlap*. Bass *et al*. [BCK03] state that, for example, the ATAM (see §4.3.2) can be applied to analyzing the software quality of a software product line. A framework could be seen as a software product line. However, note that software product lines often also provide business domain functionality, whereas a J2EE meta-framework is focused in the computing domain (see §2.4).

4.3 Software architecture evaluation

As software architecture is the level at which to identify the software quality, we further studied software architecture evaluation methods. The principle objective of software architecture evaluation is to assess the potential of the chosen architecture to deliver a system capable of fulfilling required quality requirements and to identify any potential risks [DN02]. Additionally, it is quicker and less expensive to detect and fix design errors during the initial stages of software development.

With regard to evaluating a software architecture, literature identifies two categories [DN02]: measuring techniques [ZBJ04] and questioning techniques [BKB01; KBA94; KKB98]. In general, the use of both kinds of techniques leads to the best results [ZBJ04; DN02].

4.3.1 Measuring techniques

Measuring techniques suggest quantitative measurements to be made on an architecture. They are used to answer specific questions and they address specific software qualities and, therefore, are not as broadly applicable as questioning techniques [DN02]. Examples include metrics, simulation, prototypes, and experiences. Measuring techniques are often chosen when continuous measurement information needs to be available for the architects.

With analyzing frameworks, this is less relevant. As said before, measuring techniques are time-consuming and often attribute specific. They could however function as an additional step on questioning techniques where more specific information is needed.

An example of a measuring technique is the Software Architecture Evaluation Model (SAEM). This method uses the goal-question-metric technique to define metrics that measure the software quality. This provides the architect with quantitative only information. Formalizing the software architecture can be done with the use of Architecture Description Language (ADL). However, the SAEM has not been validated yet on any software system and is highly experimental [DN02].

4.3.2 Questioning techniques

Questioning techniques generate qualitative questions to be asked of an architecture and they can be applied for any given quality attribute [DN02]. Examples of this kind of methods are SAAM [KBA94] and its successor ATAM [KKB98]. The SAAM method has been successfully used for many systems. However, it is considered that modifiability is still the quality attribute evaluated by SAAM. The ATAM method has grown out of the work on individual quality attributes. The objective of ATAM is to provide a principle way of understanding a software architecture's capability with respect to multiple competing quality attributes [KKB98]. ATAM identifies the tradeoffs made between quality attributes and evaluates if these decisions are correct with respect to the desired software quality. When an architecture, for example, is designed to use a naming server, modifiability is positively influenced because the client and server are decoupled. The client does not need to know where the server is located physically. On the other hand, the extra level of indirection introduced could have a negative impact on performance.

Analyzing the architecture, as described by the ATAM and SAAM relies on two main factors: scenarios (see §4.1) and architectural views. These views are an architectural representation of the systems' architecture which are specifically made to enable the evaluators to do architecture level impact analysis that each scenarios has on the architecture. Kruchten [Kru95] identified that many architectural diagrams attempt to capture the gist of the architecture of a system and thereby only confusing users of the diagram. They propose that the description of architecture should be organized in "4+1" views, each one addressing a specific set of concerns. The "+1" is validates the views were not in conflict with each other and together did in fact describe a system meeting its requirements. Their proposal consists of the following views:

- Logical: The elements are "*key abstractions*", which are manifested in the object-oriented world as objects or object classes.
- Process: This view addresses concurrency and distribution of functionality.

- Development: This view shows the organization of software modules, libraries, subsystems, and units of development.
- Physical: This view maps other elements onto processing and communication nodes.

In [BCK03] the authors emphasize that a lot of possible views can be found in literature that are perhaps all valuable in describing the systems' architecture. However, one has to choose the views that best help in the analysis. Nevertheless, one specific view of interest for J2EE meta-frameworks (and for J2EE applications in general) is the context view. This stems from the observation that J2EE meta-frameworks are mostly used to implement business applications, which are often part of a larger suite of systems [LRV01].

A similar approach as the ATAM is described in [LBK01; BKB01], where the authors show that the general scenarios and tactics (see § 4.1), they identified in previous work can be mapped on specific features and architectural aspects of the EJB framework. They not only validated the completeness of their scenarios and tactics, but also structurally discussed the software quality of the EJB framework. This evaluation method is based on the notion of quality attribute design primitives. Such a primitive is a collection of components and connectors that collaborate to achieve some quality attribute goal (expressed as a general scenario). In later papers and in [BCK03] the authors refer to tactics when speaking in terms off quality attribute design primitives. The proposed method is based on generating specific EJB scenarios from general scenarios. From these specific scenarios the tactics related to the scenarios are selected. The relevant tactics are subsequently mapped on EJB specific features. Then a qualitative evaluation is made on how the EJB features implement the tactics and the tradeoffs and side effects that have been made. At the end, an overview is made on how each EJB feature affects a quality attribute and therefore takes the interaction between attributes into account (just as in the ATAM). One problem with this approach is that it is not necessarily true that each general scenario can be mapped to a specific design decision or feature supported by the framework and vice versa. And the same applies to mapping framework features to general tactics. However, it can be helpful to have a list of scenarios and tactics in identifying relevant scenarios and features of the framework.

4.3.3 Remaining techniques

Another technique discussed in [DN02] is the Scenario-Based Architecture Reengineering (SBAR) method. This method uses different techniques depending on the quality attribute to evaluate. Development attributes as, maintainability and reusability, are often only evaluated with scenarios (questioning techniques). Operational software qualities as performance, reliability are first evaluated with scenarios and the output of this evaluation is then further investigated by simulation or mathematical models. This approach is therefore based on both measuring and questioning techniques dependent on the kind of attribute that needs to be evaluated. We believe that this is a good approach, because a qualitative analysis of performance only identifies risks and it then still remains hard to predict the execution.

Most of the above mentioned approaches are based on evaluating the generated specific scenarios out of the general scenarios. L. Zhu et al. [ZBJ04] recognized that collecting scenarios is not a trivial task. They identified that scenarios can be generated out of the description of patterns used in the architecture, or in our case the framework and that this helps scenarios collection. This is justified by the fact that design choices are often based on patterns. According to [DN02], one of the major purposes of using patterns is to develop software systems that are expected to provide the desired level of quality attributes. Design patterns are generally seen as a means to evaluate the qualities a framework or software system addresses. According to [DN02] patterns are a means to explain why design decisions have been made.

The authors in this paper also state that qualitative measures alone do not produce the best results and that extracting information from patterns to produce quality attribute sensitive general scenarios can help the architect to better select or develop a reasoning framework. A reasoning framework can be a prototype, a mathematical model etc. on which architectural qualities can be measured in more detail, for example the performance attribute. Creating a reasoning framework is among dependent on the need for more detailed analysis of a quality attribute.

4.4 Summary

This chapter provided the related work conducted in software quality analysis. Therefore, we first defined software quality as: "*the degree to which software possesses a desired combination of quality attributes*" (Q10). In addition, scenarios were elaborated as the means to document the non-functional requirements belonging to specific quality attributes in a consistent manner (Q11). Thereafter, it was concluded that software architecture is the appropriate level at which to determine the software quality. Subsequently software architectures for J2EE types of systems are defined and a number of software architectural analysis methods are discussed. Among the analysis methods, the questioning techniques are the most important ones in light of the remaining chapters (Q12).

Chapter 5

Framework Software Quality Analysis Method (FSQAM)

This chapter discusses the proposed method: FSQAM.

5.1 Introduction to FSQAM

The approach presented is mainly based on software architectural analysis for the reasons mentioned in the previous chapter. Throughout this document the method will be named Framework Software Quality Analysis Method (FSQAM).

FSQAM is a questioning technique (§4.3.2). Choosing this kind of technique is based on two observations. At first, measuring techniques (§4.3.1) are mostly used for specific measurements of one quality attribute, which does not provide insight into the software quality as defined in this research. By choosing a questioning technique however, there is still room for such measurements, as for example proposed in the SBAR method (see §4.3.3).

Secondly, measurement techniques are most suited for continuous measurement to also validate if development activities adhere to the specified architecture. Frameworks support qualities through the way they provide and implement participants. Naturally, developers can choose to misuse functionality offered by the framework, but that does not affect the software quality supported by the framework; it does however influence the software quality of the application build with the framework. The latter though, is out of the scope of FSQAM.

FSQAM shows the most similarity with the research conducted in [LBK01]. There the authors showed that the software quality of a framework could be discussed according to the features it provided (§4.3.2). In addition, some activities of the ATAM were used as for example identifying the high-level goals and the basic building blocks used in the framework (respectively §5.2.1 and §5.2.2).

FSQAM consists of three general phases, as shown by Figure 6.

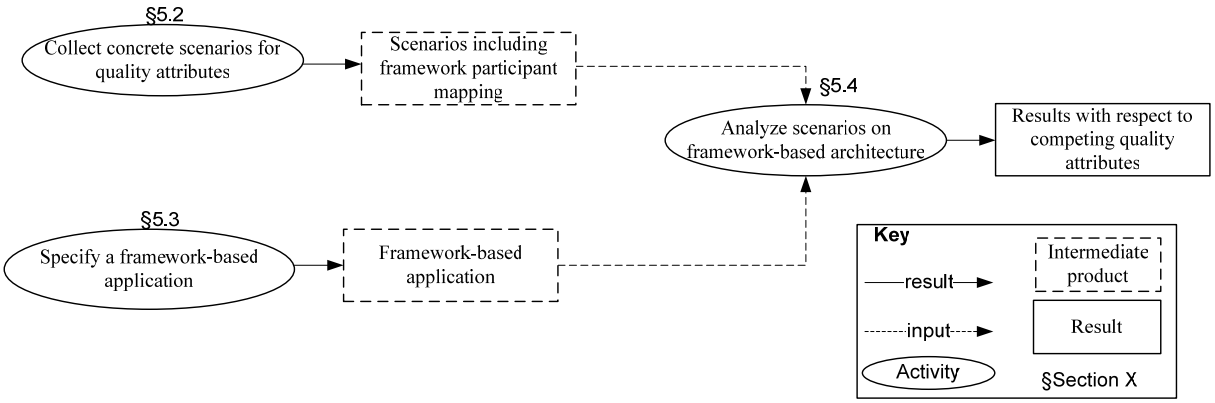

Figure 6: Overview proposed approach

The first phase consists of collecting concrete scenarios. As described in §4.1, an architecture is either suitable or unsuitable with respect to its ability to deliver particular quality attributes to the system(s) build from it. The important quality attributes of the system can be described at a high-level, as for example in §2.3 and §3.2. However, these high-level goals are not detailed enough for analysis. Modifiable in what?

Ported to which machines? To make analysis possible, concrete scenarios (see §4.1) need to be gathered. Thereafter, specific framework participants are mapped onto these scenarios, which are then described in the context of how these affect the software quality. Section 5.2 provides a more detailed explanation of this phase.

The second phase consists of specifying a framework-based application. In normal software architecture evaluations, an existing architecture is evaluated. However, frameworks are used to build different applications with different reference architectures. Based on the output of the first phase, a basic concrete application will be developed on which the collected scenarios are analyzed. Section 5.3 elaborates this phase in more detail.

Subsequently, the third phase consists of analyzing the impact of the scenarios on the framework-based architecture with respect to the quality attributes affected.

5.2 Collect concrete scenarios

Figure 7 shows the process for collecting the concrete scenarios.

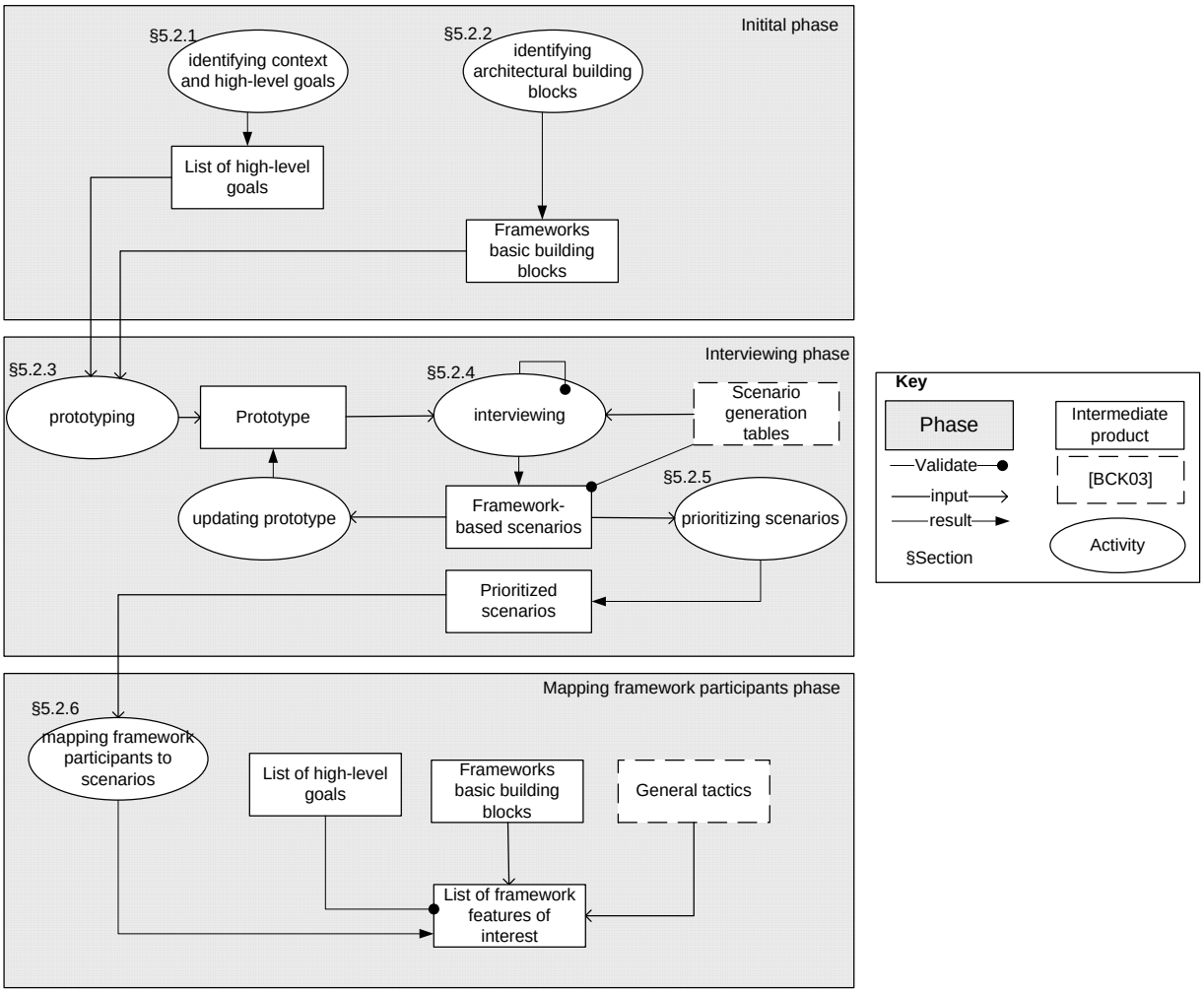

Figure 7: Approach collecting scenarios

This process is divided in three general phases for readability reasons. During the remaining document, each activity will be discussed separate of the phase it belongs to. The first phase consists of acquiring the basic knowledge of the framework that sets the foundation for the rest of the evaluation. The output of the first phase will then be used to gather scenarios. In addition, framework participants are identified that help in accomplishing those scenarios. Each activity will now be discussed in separate sections, which is also represented in Figure 7.

5.2.1 Phase 1: Identifying context and high-level goals

This activity involves getting to know the context in which the framework can be used. This should provide the evaluator with a first introduction to the framework. Then the primary goals that motivated the development of the framework must be discussed. Note that among these goals will be the ones identified in Chapter 2 and Chapter 3.

The necessity of this activity is validated by the fact that a framework, just as an architecture (see §4.3), is either suitable or unsuitable with respect to its ability to deliver particular quality attributes to the systems. Suppose that one of the high-level goals of the framework regards the development of scalable applications. If the analysis points out that mostly modifiability is addressed by the framework participants at the cost of scalability, then one can conclude that the framework is not usable for applications that need to be highly scalable.

5.2.2 Phase 2: Identifying architectural building blocks

The second activity consists of identifying the basic building blocks of the framework. That is the identification of the architectural patterns, styles and ideas that framework creators implemented in the framework. Additionally, there needs to be a description of the types of applications one can develop with the framework. The reference architecture of J2EE is discussed in §3.1. However, it could be that the J2EE meta-framework allows for different styles, such as using the framework in a J2SE environment or only in a web container.

5.2.3 Phase 3: Prototyping

Prototyping as input for the interviews serves two goals:

- Enhance interviewer/interviewee communication
- Increase interviewee's creativity

At first, frameworks that are not applied in an application are too abstract to talk about, especially when the interviewees do not have experience with the framework. By providing them and the evaluator with a representation of a real architecture based on the framework, both can speak of concrete architectural terms or issues.

With respect to the second goal, one can expect that providing the stakeholders with an application and its context that the creativity is enhanced. When using for example a web-based buy-a-bike application as the prototype, interviewees are triggered to come up with scenarios as: "*the product supplier of "Gazelle" needs to be notified when the stock for product item "335" exceeds the lower bound of five*." Subsequently, the interviewer could conclude that this scenario is not of use in the current analysis. He could however, then ask how the communication should occur, which can be of influence to a framework participant.

Prototypes are not one of the deliverables. Prototypes are only a means to accomplish a goal. Therefore it is not necessary to have a complete architectural description. One should be able to rapidly adjust the prototype. As can be seen in figure 7, the prototype can be changed for a subsequent interview based on received information. Additionally, it can even be updated during the interviews as long as the interviewee needs that information.

5.2.4 Phase 4: Interviewing

Interviewing is about eliciting what possible framework users expect that the J2EE framework has to offer and report these requirements in the form of scenarios (see §4.1). The extracted scenarios should be detailed enough for analysis, but still lie in the boundaries of the high-level goals.

Interviewing as elicitation technique is good for getting knowledge about the present work in the domain and the present problems [Lau02]. There are several types of interviews. At first, open interviews are appropriate when the interviewer has limited knowledge about a topic or wants an insider perspective [Lee02]. These kind of interviews are more like conversations and provide the interviewer with lots of information. A risk of this approach is that one wanders off in a direction that is completely out of the context and it is therefore not suited for hypothesis testing.

When the interviewer has enough knowledge about a topic and wants specific answers to specific questions, then closed interviews are appropriate. A risk, however is that the interviewer makes wrong assumptions, thereby passing over some question and missing a lot of valuable information. Additionally, the closed questions are often repeated during different interviews to determine agreement between different stakeholders. However, as discussed in §4.3.2 and in [BCK03], different stakeholders have different needs in an architecture. Therefore, in an ideal situation, interviews should be conducted with representatives of each stakeholder group.

Semi structured interviews are in the middle of both. These kind of interviews start off with open questions and try to get to the bottom of things by asking related more closed questions [Lee02]. These kind of interviews are best suited for scenario extraction, because it leaves room for discussion and the interviewer can beforehand determine the topics that need to be discussed.

As shown in Figure 7, input for the interviews consists of prototypes and scenario generation tables (§5.2.3, Second Addendum). The latter is useful in two ways. At first, they can be used to direct the interview to make general scenarios concrete for the prototype. Secondly, after interviewing they provide their use as a form of validation of which the outcome can be used to direct the next interview through a new prototype or other questions.

5.2.5 Phase 5: Prioritizing scenarios

The collected scenarios in the previous phase can easily contain more scenarios to analyze then there is time for. Therefore, the scenarios need to be prioritized. There are a lot of prioritizing techniques and the one best known to the evaluator and the stakeholders should be used. The output of this phase is a prioritized list of scenarios that are used to plan the remainder of the analysis. It should tell the evaluator where to spend its (relatively limited) time [BCK03].

The scenarios are prioritized before mapping of framework features occurs. This choice is motivated by the fact that the next activity can take an extensive amount of time. Up to this point the evaluator only has a fairly limited knowledge of the framework as a result of the second activity (§5.2.2). Determining which framework participants have an impact on which scenarios requires much more research into the framework. By firstly selecting the important scenarios through this activity, the evaluators can subsequently spent more time on the actual analysis. If the evaluators have any time left after the analysis of the highly-prioritized scenarios, then they could analyse the next set of highly-prioritized scenarios.

5.2.6 Phase 6: Mapping framework participants to scenarios

This activity selects the scenarios from the set of highly-prioritized scenarios that lend themselves for analysis. The result of this phase is a set of scenarios for which the framework offers participants that help in accomplishing that scenario. The result of this activity is the mapping of participants to scenarios and a categorization in of the scenarios as follows:

- *Non-relevant, not supported.* The framework should not provide participants for this scenario and actually does not. Examples include: "*the system should be adapted to provide undo functionality.*" Most of these scenarios will already be filtered during the interviews (§5.2.4).
- *Non-relevant, supported.* The framework should not provide participants for this scenario, but it does. The participants that help in accomplishing this scenario are outside of the scope of the framework and do not contribute to a framework's high-level goal: "*Frameworks have to achieve its primary purpose*" (§2.3).
- *Relevant, not supported*. The framework should provide participants for this scenario but does not. The participants that are missing do not aid in the framework's high-level goal: "*increase in developer's productivity*" (§2.3).
- *Relevant, supported*. The framework should provide participants for this scenario and does. This category is analyzed in the next activities.

The general tactics, discussed in §4.1, provide a means to identify framework participants. For example, the general tactic "*caching*" can be mapped onto how caching is provided by the framework. In §4.3.2 we already discussed a research that was able to map the general tactics to EJB specific participants [LBK01] and additionally analyze the software quality. Besides the tactics, the list of architectural building blocks (§5.2.2) can also be used in the mapping process.

5.3 Specify a framework-based application

The framework-based application is a concrete specific instance of all the applications that could be developed with the framework. Determining the architecture of the application is based on:

- The output of the prototyping activity.
- The output of the mapping activity.

At first, during the prototyping phase, one or more prototypes are developed which is preferably reused in this activity. In addition, the application should be suited for analysis of the selected scenarios. For operational scenarios this entails measuring response times and in case of development scenarios it includes applying modifications to the application.

This activity basically consists of implementing and documenting the application. The former will ensure that the evaluator gains experience in using the framework, which is necessary for the actual analysis phase.

The addition in experience should also be used to determine the effect of the framework on the ease of development. If the framework relies on a technique that prevents proper execution of the refactoring capabilities of IDE's, then this could decrease the developer's productivity.

Documenting the application contributes to the discussion of the framework participants (§5.4) and can be done with the use of architectural views (§4.3.2). The result of this phase is a documented application built on top of the framework.

5.4 Analyze scenarios on framework-based application

The last phase consists of executing the scenarios onto the framework-based architecture. The framework-based architectural views can play a supporting role. However, when the scenarios are also implemented, the following benefits are gained:

- Validation of the mapping of framework participants for: incorrect mapping and missing framework participants.
- Addition in experience with the framework, by which the evaluator can also analyse the impact of the framework on the ease of development.
- The resulting application can be reused as a reference application by others to implement specific scenarios.

For each scenario, the impact on the architecture by the framework participants needs to be discussed. Note that the participants are already identified in §5.2.6. In addition, the software quality attribute ramification for each participant needs to be deternined. If participants help in accomplishing one or more scenarios then a reference is made to the scenario that already describes the participant. The template that can be used for each participant consists of the following four items:

- Quality attribute analysis
- Negative side effects
- Trade-off analysis
- Quality attribute summary

The quality attribute analysis describes how and which quality attributes are positively addressed. Subsequently, the negative side effects are described; that is the quality attributes that are negatively affected by this participant. The trade-off analysis then provides a rule of thumb regarding when to or how to apply this participant. The last item is a quality attribute summary. In that summary all quality attributes receive a "–" (negatively addressed), "0" (not addressed) or "+" (positively addressed) indicating how each attribute is addressed by the participant. This notation should only be used to present an abstract overview of the software quality supported by that participant. One should note that without any context, provided in the previous items, the latter is meaningless.

Finally, a summary is provided in which the high-level goals, determined in the first activity of the collecting phase, are evaluated to the results of the analysis of independent framework participants. In addition these results are presented to the stakeholders of the analysis.

5.5 Summary

This chapter proposed a method for analyzing the software quality of a J2EE meta-framework. First, we provided an introduction to FSQAM that among discussed some key decisions as the choice for a questioning technique and the resemblance of FSQAM to the research discussed in [BCK03]. Subsequently, we shortly discussed the purpose of each of the three phases of FSQAM (§5.1): 1) "*collect concrete scenarios for quality attribute*s" 2) "*specify a framework-based application*" and 3) "*analyze scenarios on framework-based application"*. The result of executing these phases is a document that describes too which degree the software quality of the framework corresponds to the high-level goals determined in the activity "*identifying the context and high-level goals"* (§5.2.1).

The next chapter will discuss the application of FSQAM to the Spring framework.

Chapter 6

A case study: Spring

This chapter provides the application of FSQAM to a J2EE meta-framework named Spring [JHA06].Each phase presented in Figure 6 will be discussed in section 6.1 through 6.3. The result will be a summary of the software quality supported by the Spring framework (§6.3.10).

6.1 Collect concrete scenarios

This section will discuss the execution of the first phase of FSQAM. The next sub sections discusses how each activity, shown in Figure 7, is carried out and what the results are.

6.1.1 Phase 1: Identifying context and high-level goals

The context and business drivers presented in this section are based on studying Spring's documentation [WB05; JHA05; JHA06]. The context and backgrounds of Spring are already discussed in §1.1 and therefore only the high-level goals of this framework are presented. Extracting these goals not only stem from Spring's documentation. At Info Support, each intern has a customer which is one of the company's senior developers. During the first meetings with this customer (ing. B. Rodenburg) we discussed the goals a meta-framework should set. Additionally, we used the goals found during the literature study (§2.3, §2.5 and §3.2).

From these three sources we identified the following high-level goals for J2EE meta-framework:

HG-1) *Simplify and increase productivity of developing J2EE applications.* Traditional J2EE programming is considered complex even in the case of a simple web application. It is Spring's pledge to deliver J2EE services (transactions, state management, multi-threading, security and resource pooling) in a way that the complexity of the application is proportional to the complexity of the problem.

HG-2) *Increase modifiability of J2EE framework-based applications.* The Spring framework should promote separation of concerns and it preferably provides infrastructure instead of generation of code. The traditional J2EE specification also sets the goal that the applications should be portable across different application server vendors. However, in practice this is very hard to achieve. Many web applications also do not need to run on expensive application servers, but are better of running on a web container. Spring tries to achieve that the application code can run without Spring or any container.

HG-3) *Facilitate in testing and debugging activities*. It is difficult to test applications based on EJB and other traditional J2EE technologies without an application server. Yet unit testing outside an application server is essential in achieving high test coverage and reproduction of failure scenarios. Additionally, the start up time of the container also decreases the motivation for developers to continuously run the unit tests.

HG-4) *Foster integration with other frameworks.* There are all kinds of good tested solutions, which should be supported by Spring instead of redeveloped. Besides, it should be possible to use Spring functionality within other frameworks.

HG-5) *Minimize learning curve.* It is Spring's aim to provide a consistent programming model, so that its users quickly learn to adapt all framework's hooks to their needs. Another aim is to offer good documentation, supporting IDE's, community support and getting large vendors to support Spring.

HG-6) *Provide support for remote communication interoperation.* The framework should support various ways for remote communication interoperation across different vendor implementations.

HG-7) *Provide effective and reliable operation at many different scales.* The framework should facilitate the effective operation of the application at many different scales even despite the occurrence of failures.

HG-8) *Provide efficient and predictable execution.* The framework should enable the efficient and predictable execution of the applications built on top of it.

The high-level goals identified are represented by “HG” followed by a number, so that they can be referenced throughout the remaining document:

Another aspect discussed with the customer was the scope of this case study. At first, we decided that the presentation tier (Figure 5) should only be addressed after analysis of the server-side. Secondly, there was determined that the following quality attributes are of interest in this analysis: availability, modifiability, performance, testability and usability. The Second Addendum provides, for each quality attribute, a definition and the scenario generation table (§4.1). In consideration with our customer, we decided to leave out security because this domain would take to much time for the evaluator to become familiar with. Secondly, we decided that usability (for which by our knowledge no generation table exists) should refer to the ease with which developers can develop when using the framework. This encompasses issues as documentation and if all capabilities of IDE’s, when developing with the framework, are still available. As concluded in section 2.3, frameworks should increase the developer’s productivity. However, if such features are not an option when developing with the framework then this could be at the cost of the increase in productivity.

6.1.2 Phase 2: Identifying architectural building blocks

Spring is based on the following three fundamental building blocks [WB05; JHA05; JHA06]:

- Dependency injection[6]
- Spring Aspect Oriented Programming (AOP)[6]
- Service abstraction layers[6]

These three building blocks are now shortly examined, for a more elaborate discussion see the Third Addendum.

Dependency injection is a means to wire different elements together. The basic idea is that the Spring container functions as an assembler that populates a field in a class that is dependent on this field for its correct execution.

Spring AOP (and AOP in general) provides a different way of thinking about code structure compared to OO or procedural programming. AOP enables us to think about concerns or aspects in the system. Concerns as logging, transaction management and failure monitoring are often in numerous classes throughout the architecture. The problem with traditional techniques is one is not able to separate these concerns. AOP addresses this by providing an insertion mechanism to apply aspects, such as logging, to add aspect code to designated locations in the base code.

The third building block, the service abstraction layers, addresses a disadvantage of J2EE API’s. In case of infrastructural issues as connecting to a persistence store, there is no way to avoid working with an API. The J2EE APIs are mostly initially written for usage behind the scenes. JTA, for example, was intended for use in EJB declarative transaction management. Using JTA directly becomes cumbersome because of having multiple exceptions that each need their own catch block. This means a lot of code throughout your application logic which is abstracted away by Spring APIs.

These three framework participants accomplish a Plain Old Java Object[7] (POJO) programming model, in which POJOs do not need to call enterprise services explicitly, but do so by accessing well-defined interfaces with no implicit dependencies.

With respect to the architecture of an application built on top of Spring there can be many variations. The Spring’s architecture is modular, which should provide developers with the benefit that one can only use a specific Spring participant. For example, when using dependency injection one is not obligated to use Spring AOP or one of the abstraction service layers. This model also ensures that Spring cannot only be used in a full-blown application server, but some features, as dependency injection and Spring AOP, in a J2SE environment.

6.1.3 Phase 3: Prototyping

During this activity a web-shop for buying bikes was used as the prototype. The choice for this prototype was fairly obvious because most employees at Info Support are known with this case.

The first choice we made was to start with a most basic application. Then, during the interviews, the

[6] Third Addendum Spring’s basic building blocks
[7] http://en.wikipedia.org/wiki/Plain_Old_Java_Object

interviewees could use their field experience to come up with scenarios that make the application more usable for real life use. This is also a useful approach to validate if Spring's complexity is proportional to the complexity of the problem (HG-1). The domain used in the application should also be as small as possible, because more domain functionality only increases complexity while it does not aid in gathering more or better scenarios that describe the non-functional requirements.

A second explicit decision was that it should be easy to change the documentation after and during the interviews. We therefore not digitally produced the views, but created the figures before or during the interview. In a number of cases, we also asked interviewees to draw what they meant into a view. These figures are not represented here. However, the specified framework-based architecture (§6.2) is mostly the result of this activity and will be discussed in §6.2.

6.1.4 Phase 4: Interviewing

As discussed in §6.1.1, our analysis is mostly conducted in consideration with ing. B. Rodenburg. Several interviews, at the start of the project, were conducted with this senior developer. These interviews were typical open interviews (see §5.2.4) and had the following goals:

- Getting to know each other and feel comfortable.
- Acquiring his main concerns and questions regarding the Spring framework (expressed as high-level goals in §6.1.1)
- Setting the scope and quality attributes of this case study (§6.1.1).
- Increasing the understanding and issues in J2EE development of the present author.

As expected, these interviews increased the understanding and issues in J2EE development. It also multiple times changed our prototype. According to that prototype and a list of questions (among extracted from the scenario generation tables) an additional four interviews were conducted with respectively two senior developers, an IT architect and a trainer at Info Support's knowledge centre.

As stated in §5.2.4, interviews should preferably be held with representatives of each stakeholder group. However, despite Info Support's best efforts, not all representatives were at that time available. This issue is addressed in two ways:

- Interviewed stakeholders have different expertise.
- Validation of the found scenarios by the high-level goals and scenario generation tables.

At first, the interviewed stakeholders have different expertise in different areas. The IT architect was primarily focused on key architectural decisions related to performance and scalability. One of the senior developers had extensive experience in unit testing. The other developer was mostly concerned about how the framework would affect the possibilities of development environments. Secondly, after each interview we determined if the found scenarios could be tracked to one or more of the high-level goals or the other way around. If a high-level goal was not addressed, then the next interview could be directed towards it. This same process was conducted for the scenario generation tables, which then functioned as an indication of possible interesting scenarios not yet collected.

The interviews were semi structured. At the start of the interview a first introduction into the goals of that interview was provided by the interviewer. This was followed by a number of open questions. During the interviews, we continually wondered if the direction of the interview led to valuable scenarios. If not, then the discussion was directed. For instance, scenarios which describe that an "*Order*" object should be mocked for testing purposes is similar to a scenario that described mocking a "*Customer*" object. The framework will not provide different support for a "*Customer*" and an "*Order*". Another example, are scenarios that describe an addition of functionality in the domain or a change in the colour of the user interface. It was the purpose of the interview to find as much diverse scenarios that address non-functional requirements.

Each interview resulted in a new set of scenarios and a modified question list. The reasons for changing the question list were threefold: if a question was answered the same twice, it was altered or removed to fully utilize the precious time available during the next interview. Secondly, interviews lead to new questions that were added to the list. Finally, different interviewees have different expertise which should be fully used (as discussed above).

During the interviews the prototype proved its value. For instance, during one of the conversations, the initial open-ended question was: *"Can you name a couple of architectural decisions you make regarding the implementation of business services?*" The interviewee named a couple that were already gathered and then came up with caching. Additionally, the following question was asked: "*Considering the web shop application, where do you think caching should be applied and why?*". By asking this question, the interviewee provided the context in which caching should be used which made it better understandable for the interviewer. This instance is one of the aspects the prototype proved its function. As can be concluded

from the previous example, we did not only brainstorm directly about scenarios. Asking direct questions could lead to interesting answers. After the interviews, this kind of information was documented in the form scenarios.

The result of this activity is a number of scenarios which are listed in the Fourth Addendum. The table indicated with "*MO-3 Integration with different systems*" shows one of the scenarios. Each scenario is uniquely identified by two letters that indicate the quality attribute primarily affected. Subsequently, a number is used to represent a subcategory within the quality attribute classification. This subcategory is determined according to a common aspect. For instance, modifiability scenarios are related to the common modification that must be made. When applying this theory to the example below, we indicate that this scenario is a modifiability scenario (MO), it is in the third subcategory (MO-3 Integration with different systems) and it is the first scenario in this group.

MO-3. Integration with different systems		
Nr.	Spring specific scenarios	Priority
MO-3.1	The developer wishes that an external .NET marketing system can place customers through the customer service by use of web services. The change should be made at design time and be functional and tested within eight hours.	(M, M)

6.1.5 Phase 5: Prioritizing scenarios

Prioritizing was only executed in cooperation with our customer, ing. Rodenburg with use of the MOSCOW rules [Lar02], which is the abbreviation for "*Must have*", "*Should have*", "*Could have*" and "*Would like to have but won't have this time around*". In addition to this prioritization, the present author has prioritized the scenarios according to the amount of time he believes implementing and analyzing the scenario would take. Estimating the time it would take is difficult because of the lack of knowledge regarding the framework's abilities. This estimation was presented as follows:

- Very Short (VS) which represents one day.
- Short (S) which represents two days.
- Long (L) which represents three days.
- Very Long (VL) which represents one week.

The representation has been kept simple and was only intended to separate the small and straightforward scenarios from the large and complicated ones. This prioritization can be viewed in the Fourth Addendum. Considering the time available we decided to analyze the pairs indicated by (M, VL) which stands for "*Must have, Very Long*". However, these were not available and we therefore decided to analyze the (M, L) pairs.

The following table shows the scenarios that are further analyzed in the next activity.

Table 2: Selected scenarios after prioritizing		
Nr.	Spring specific scenario	Priority
AV-1.1	The MySQL database server fails to respond. The system identifies the failure, logs it, notifies the administrator by mail and switches to the redundant database. The system will continue to provide its services with a maximum delay of five seconds.	(M,L)
MO-1.2	The developer wishes to change from Hibernate to IBatis. The ORM mapper is replaced, tested and functional within three days at compile time.	(M, L)
MO-1.4	The developer wishes to add logging to all existing methods in "CustomerServiceImpl" and "OrderServiceImpl". The functionality is added at design time.	(M, L)
MO-3.1	The developer wishes that an external .NET marketing system can place customers through the customer service by use of web services. The change should be made at design time.	(M, L)
MO-3.2	The developer wishes to set up communication with an external payment system so that customers can pay their orders. This system provides its services through exposing a web service. The functionality is added and tested at design time.	(M, L)
MO-3.10	The developer wishes that when adding a customer to the database, it is also added to an external J2EE marketing system. The system is changed and tested at design time. This system provides its services through the use of a *stateless session bean* that runs in an IBM Websphere application server 5.1. The system is changed and tested at design time.	(M, L)
TB-2.2	The developer wishes to be able to test the DAOs without actually modifying the data in the data store. The DAOs are tested with the use of the database, but the data in the database is unchanged.	(M, L)

6.1.6 Phase 6: Mapping framework participants to scenarios

This activity verifies whether the framework provides participants that support the implementation of the selected scenarios. For instance, a more detailed study to Spring's documentation learned that it provides an abstraction layer over the traditional JavaMail API. It thereby helps in accomplishing scenario AV-1.1 (Table 2) in which the system administrator is notified by mail when the database server fails.

The division of the above scenarios in the four categories mentioned in section 5.2.6 proved to be unnecessary. We actually only identified scenarios for which the framework supports participants.

During this activity we also experienced that it is useful to verify if the set of selected scenarios address the high-level goals. With the current selected set, we do not completely verify high-level goal "*HG-4) foster integration with other frameworks*". This is because Spring is not used within the control flow of another framework. We, in cooperation with ing. Rodenburg, decided that the following scenario should be added to the list.

MO-3.7	The developer wishes that an external J2EE marketing system's client side, which is build with the Eclipse Rich Client framework, can communicate with Spring bean services via *HTTP Invoker*. The system is changed and tested at design time.	(M, L)

6.2 Specify a framework-based application

The application used in this research is a web-based buy-a-bike application. This application provides all kinds of functionality which is described in [ISE06]. Regarding all this functionality we observed that we would not need it all in order to analyze the scenarios. Subsequently, we decided that our application would consists of just two domain objects, a customer and an order. The web-based application should provide users with CRUD (Create, Read, Update, Delete) functionality for both the customer and the order. In addition, an order can only be created for a specific customer.

The following sub-sections will chronologically discuss the logical and physical view. We decided that the context view as discussed earlier could be used in combination with the physical view without having the negative effects described in section §4.3.2.

6.2.1 Logical view

Figure 8 shows the logical view of the buy-a-bike application. The layered architecture left of the dotted line shows the architecture we started with, and to the right shows the replacement of the persistency layer (as described in scenario AV-1.1). We have chosen a three-tiered model because this is most often used as specified by our stakeholders. The presentation layer consists of eight controllers according to the Model View Controller architecture for web applications[8]. Each controller is wired with the services it needs through dependency injection (see the Third Addendum for an example). The services in the business layer are also wired with their needed dependencies, either "*CustomerHibernateDaoImpl*", "*OrderHibernateDaoImpl*" or both, by dependency injection. In this case, the basic building block dependency injection couples the layers together. The primary benefit gained is that they are loosely coupled, thereby addressing modifiability and testability. For instance, specifying that "*CustomerServiceImpl*" needs to use another DAO object is accomplished in configuration, which is also useful for mocking purposes when unit testing.

6.2.2 Physical view

This view shows with which other systems the buy-bike-application communicates. Note that this view represents the situation after implementing the scenarios. The situation before implementation only consisted of our application executing in the tomcat web container and MySQL running on this same physical machine.

The selected scenarios for analysis are shown in this figure. For instance, the IBM Websphere server executes the J2EE marketing system that publishes functionality through web services. In the following section we will subsequently discuss how our application needs to be adjusted in order to communicate with this web service. In the same way the discussion of the other scenarios are supported by these views.

[8] http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html

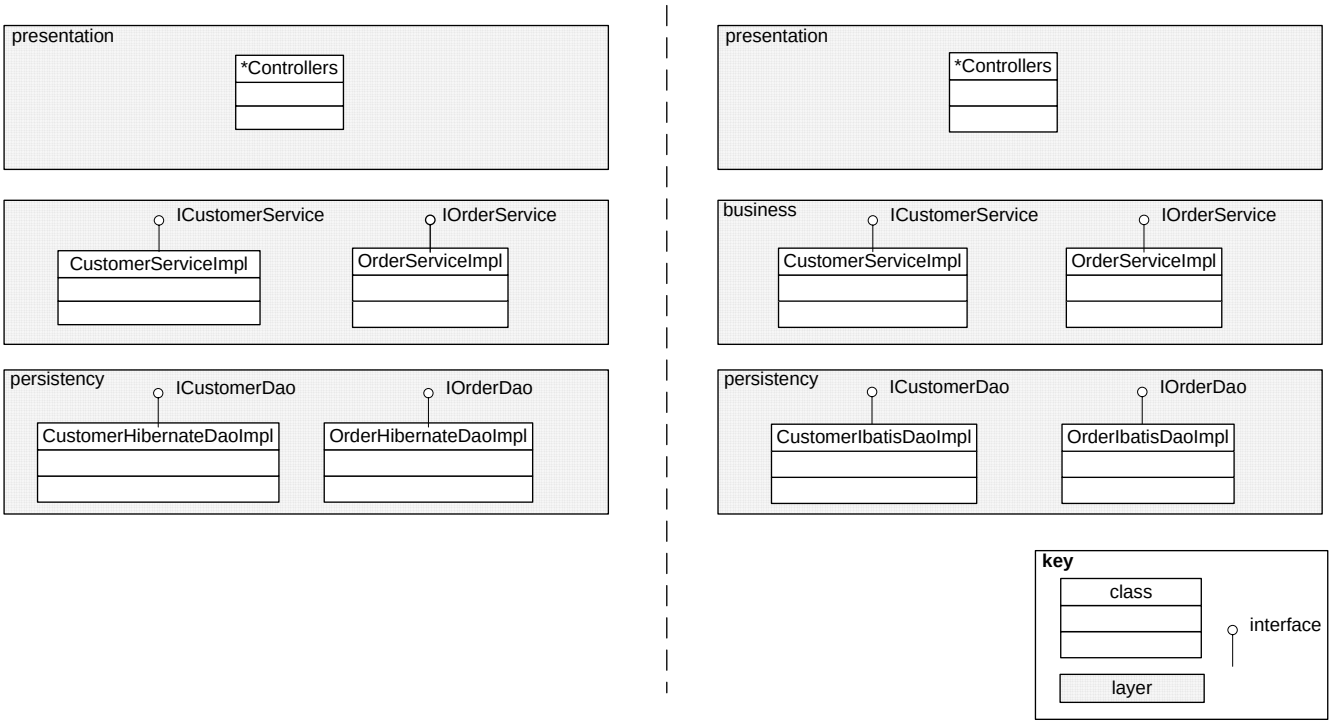

Figure 8: Logical view buy-a-bike

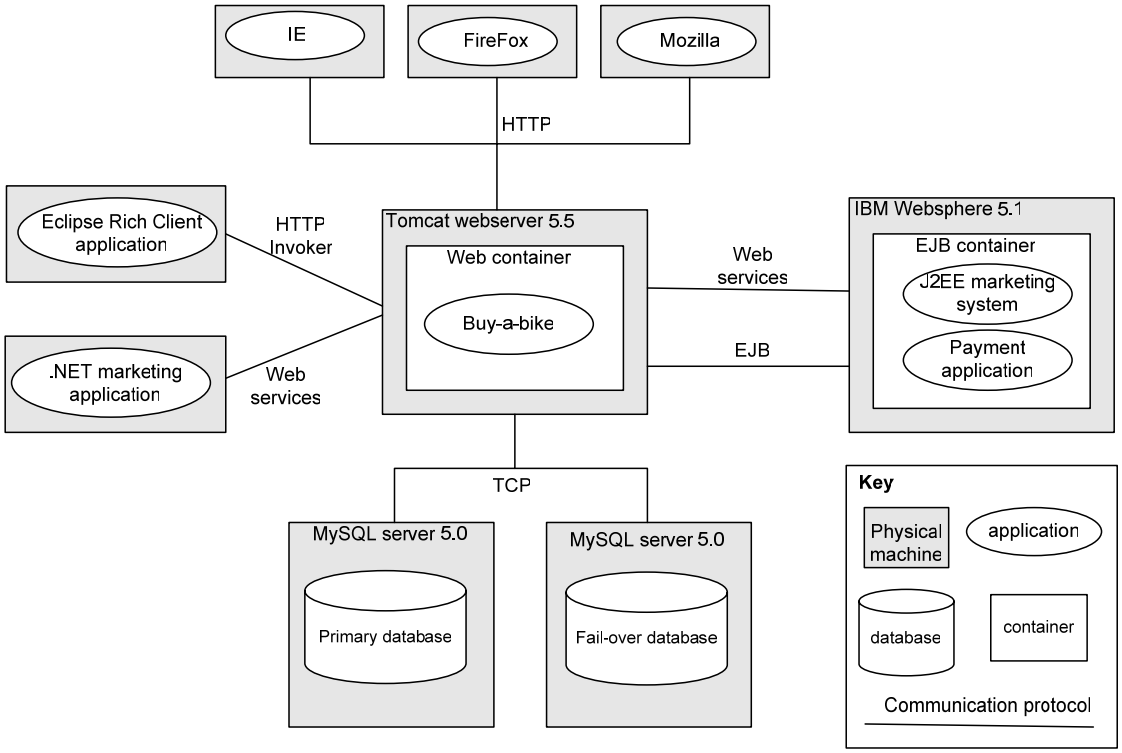

Figure 9: Physical view buy-a-bike

6.3 Analyze scenarios on framework-based application

This paragraph discusses the last activity of our process (shown in Figure 6). Each scenario will be discussed according to the template described in §5.4. The Third Addendum explains the basic building blocks of Spring which must be understood in order to understand this section.

6.3.1 AV-1.1 MySQL database server failover

Before elaborating how this scenario is implemented with use of the Spring framework there are two things to note. At first, keeping the failover database in synchronization with the primary one is out of the scope of this scenario. Secondly, in section 6.2 we provided a logical view in which we showed the Hibernate based DAOs. These DAOs all rely on a datasource for communication with a database. In the following text, we refer to these DAOs as one object named "*HibernateBackend*".

Code fragment 1 provides the configuration code for this failover scenario. The "*HibernateBackend*" is a class that communicates with the database though a "*datasource*" object. However, the object does not directly receive an instance of type "*Datasource*", but a bean of type "*ProxyFactoryBean*". The latter is a Spring participant that is primarily responsible for the creation of proxies.

```
<bean id="primaryDS" class="DataSource">
        <property name="url" value="primary"/>
</bean>

<bean id="secondaryDS" class="DataSource">
        <property name="url" value="secondary"/>
</bean>

<bean id="swappingDatasource" class="org.springframework.aop.target.HotSwappableTargetSource">
         <constructor -arg ref="primaryDS"/>
</bean>

<bean id="dataSource" class=" org.springframework.aop.target.ProxyFactoryBean">
        <property name="targetSource" ref=" swappingDatasource"/>
        <property name="interceptorNames" ref=" swapDataSourceAdvice"/>
</bean>

<bean class=" HibernateBackend ">
        <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="swapDataSourceAdvice" class="SwapDataSourceAdvice">
        <property name="primaryDS" ref="primaryDS"/>
        <property name="secondaryDS" ref="secondaryDS">
        <property name="targetSource" ref="swappingDatasource">
</bean>
```

Code fragment 1: Configuring IBatisBackend with failover

A property of "*ProxyFactoryBean*" is the "*targetResource*" that can be set to determine the target object of the proxy, which is in this case the "*HotSwappableTargetSource*" participant. The role of the latter Spring class is to enable a switch of the target object at runtime in a threadsafe manner for singleton only objects. In the above configuration this class is initialized with a reference to the "*primaryDS*" bean, therefore all calls from "*HibernateBackend*" are delegated to the primary data source.

This configuration enables one to develop an aspect that is triggered by an exception that represents database communication failure. This aspect will then call the "*swap*" method of "*HotSwappableTargetSource*" with "*secondaryDS*". Subsequent calls from "*HibernateBackend*" to the datasource will then occur on the "*secondaryDS*" bean. The code of this aspect is represented in Code fragment 2. Note that dependency injection is used to wire together the dependencies of this class. Also take notice of how the aspect is added to the "*dataSource*" bean. By setting the "*interceptorNames*" property to our aspect it is added to a chain of interceptors. At the time an exception is thrown, our advice is called.

With respect to sending an email, Spring provides a service abstraction layer on top the traditional J2EE mail API through the following two participants: "*MailSender*" and "*SimpleMailMessage*". The first interface describes two methods for sending mail. Both methods expect an object of type "*SimpleMailMessage*". The latter participant can be configured to contain properties as "*from email address*", "*to email address*" and the remaining properties one would expect for sending an email. The configuration of both is not added to Code fragment 1, because it is accomplished as in the previous examples.

```
class SwapDataSourceAdvice implements ThrowsAdvice {

	private boolean primaryDSInUse = true;
	private DataSource primaryDS;
	private DataSource secondaryDS;
	private MailSender mailSender;
	private HotSwappableTargetSource targetSource;
	private SimpleMailMessage message;

	public void setPrimaryDS(DataSource primaryDS) {
		this.primaryDS = primaryDS;
	}

	public void setSecondaryDS(DataSource secondaryDS) {
		this.secondaryDS = secondaryDS;
	}

	public void setMailSender(MailSender mailSender) {
		this.mailSender = mailSender;
	}

	public void setTargetSource(HotSwappableTargetSource targetSource) {
		this.targetSource = targetSource;
	}

	public void setMessage(SimpleMailMessage message) {
		this.message = message;
	}

	void afterThrowing(Throwable exception) {
		if (exception instanceof ConnectException) {
			if (primaryDSInUse) {
				targetSource.swap(secondaryDS);
			} else {
				targetSource.swap(primaryDS);
			}
			primaryDSInUse = !primaryDSInUse;
			mailSender.send(message);
		}
	}
}
```

Code fragment 2: SwapDataSourceAdvice implementation

The following tables will subsequently discuss the impact on quality attributes of "*ProxyFactoryBean*", "*HotSwappableTargetSource*" and the "*mail service abstraction layer*".

Analysis "*ProxyFactoryBean*" participant	
Quality attribute analysis	The "*ProxyFactoryBean*" creates a proxy based on a target object. The purpose of this proxy is to add functionality to the object being proxied. This participant represents the core concepts of Spring AOP. It affects different quality attributes in different ways. At first, modifiability is positively enhanced because: - It implements the AOP alliance, which provides a couple of interfaces specifying the signatures required to implement method interception code that can run in multiple AOP frameworks. Thereby allowing AOP code to be portable to other AOP frameworks that also implement these interfaces. - The aspects can function in any environment including application servers in contradiction to class loading typed Aspect frameworks [JH04], because they interfere with the class loading of application servers. This is also beneficial regarding the deployment process, which is exactly as one of a J2SE application. - The target POJO remains unchanged. No code needs to be generated, only infrastructure is provided (§2.3). Besides the positive effect on modifiability, testability is also increased because one can now unit test the application code in isolation of the aspect. Aspects often contain technical concerns as performance measurements, transactions etc. When intermingling the code, such

	as in the traditional approach, unit tests need to account for it. Thereby introducing lots of duplicate unit tests for different classes. This will especially be the case for development teams that use code coverage tools to monitor their coverage.
Negative side effects	The extra level of indirection introduced at runtime by the proxy, negatively affects performance. Modifiability is also addressed negatively. If objects that are advised, call methods on itself, then these methods will not be advised because the invocations do not occur through the proxy. Imagine for instance the following change request. A class contains a number of fine-grained methods that are all being advised. Subsequently, one wants to add a less fine-grained method that calls the former ones, because network calls between client and server need to be reduced. When changing this code, one could easily forget to advise the new method. Especially when some form of autoweaving (§6.3.3) is used.
Trade-off analysis	The impact of Spring AOP on performance is considerable when applying it extensively [Joh04]. For instance, when applying AOP for fine-grained persistency objects of which hundreds are created during a business transaction. However, in [Joh04] the authors performed benchmark tests, which demonstrated that using Spring AOP for stateless business POJOs performed well. We recommend starting with Spring AOP and if performance benchmarks indicate that it becomes a performance bottleneck, then one can use Spring's AspectJ service abstraction layer. According to [Joh04], this provides a considerable performance boost, because it relies on code generation. With respect to modifiability, we believe that the advantages are much greater and more often needed than the particular scenario mentioned before. We address it as a known risk which needs to be taken in account when applying Spring AOP.
QA summary	Availability 0 Modifiability + Performance - Testability + Usability -

Analysis "*HotSwappableTargetSource*" participant	
Quality attribute analysis	The use of "*HotSwappableTargetSource*" takes away the burden of coding infrastructural and threading issues and to have this code tangled across the code that actually implements business functionality. It enhances usability, because of the predefined swapping technologies.
Negative side effects	The extra level of indirection introduced at runtime by the proxy, negatively affects performance.
Trade-off analysis	The performance overhead can be considered negligible, considering that the proxy is a shared instance and just delegates the calls to the datasource currently in use.
QA summary	Availability + Modifiability 0 Performance 0 Testability 0 Usability +

Analysis "*Mail service abstraction layer*" participant	
Quality attribute analysis	The use of Spring's "*MailSender*" interface in combination with "*SimpleMailMessage*" decouples the application code from the actual mail implementation being used, which shields the user from the specifics of the underlying mail system. This separation of interface positively affects modifiability because different mail systems can be switched more easily and only in configuration. Secondly, the implementations of the "*MailSender*" interface provided by Spring improves usability because it abstracts away implementation specific code. Another benefit regards testability. In our experience testing classes that rely on the JavaMail API are hard to test, because of its final classes and system properties. Using the "*MailSender*" interface makes mocking easier and thereby unit testing.
Negative side effects	The extra layer of indirection again negatively influences performance. It also masks certain features of a specific implementation, thereby limiting the developer to the functionality offered by the Spring participants.
Trade-off analysis	The performance impact can be considered negligible. The Spring service abstraction layer provides enough support for the most common mail sending scenarios. If developers wish to use specific mail sending features of their mail sending system, they should use that API.
QA summary	Availability 0 Modifiability + Performance - Testability + Usability +

6.3.2 MO-1.2 Change of Object Relational Mapper

As can be seen in the development view of the framework-based architecture (§6.2), the application currently uses Hibernate[9] DAOs for its persistency. The layered architecture, the use of dependency injection and the DAO pattern should provide the means to be able to replace the persistency layer by an IBatis[10] based persistency layer only through configuration, without any ripple effects to the business logic. Before wiring in the new IBatis based DAOs, they need to be created.

Spring provides two service abstraction participants for the creation of an IBatis DAO: "*SqlMapClientTemplate*" and "*SqlMapClientDaoSupport*". The main difference between these two is that the former is injected into - and the latter is extended by clients. Considering the discussion in §2.2 between white- and black box, we have chosen for the black box approach. Using the white box based extension hook is actually only necessary in more advanced situations where the interaction sequences within the framework need to be influenced [JHA06].

Code fragment 3 shows a part of the implementation of "*CustomerIBatisDaoImpl*". The template provides all kinds of methods related to persistency. When invoking such a method, an SQL query defined in an IBatis specific configuration file is executed. This is represented by the first parameter of the "*insert*" method. The second parameter of the "*insert*" method represent the "*Customer*" object that needs to be persisted.

```
private SqlMapClientTemplate ibatisTemplate;

public Customer getCustomer(int id) {
        Customer cust =
                (Customer)ibatisTemplate.queryForObject("getCustomerById", id);
        return cust;
}

public void addCustomer(Customer newCustomer) {
        ibatisTemplate.insert("addCustomer", newCustomer);
}
```

Code fragment 3: Implementation CustomerIBatisDaoImpl

After implementing the DAOs, we need to set the properties of "*SqlMapClientTemplate*". This configured bean is then wired into the "*CustomerIBatisDaoImpl*", which is shown by Code fragment 4

The "*sqlMapClient*" bean is a factory that creates and returns an instance of the IBatis "*SqlMapClient*" class, based on the location of a configuration file and a data source. Thereafter, this bean is wired into the "*sqlMapClientTemplate*".

Now that the new DAOs are created we need to configure our application so that the business services rely on the new DAOs. Spring supports this configuration by providing two participants: separate configuration files and autowiring.

Separate configuration files ensures that not all bean definitions need to be in one and the same configuration files. At start-up of the container, several XML files can be defined in which beans are declared that are grouped to a common factor. For instance, all business beans can be configured in business-layer.xml and all data related beans in persistency.xml. The new beans configured above are defined in a new configuration file called persistency-ibatis.xml.

The second participant to discuss is autowiring. Autowiring lets the Spring container automatically find the dependencies of a bean. This means that we do not have to explicitly configure that the "*customerDao*" bean has a reference to "*sqlMapClientTemplate*". The container supports the following forms of autowiring: by name, by type and auto-detection. In this scenario we set the container to autowiring by type for the business services only. This means that if the container tries to instantiate a business service object, it will search for beans in the configuration file(s) that match with the types of the properties of that business service object. If the container finds two instances of a certain type, it will not start-up and instead throw an exception.

[9] http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf
[10] http://ibatis.apache.org/

```
<bean id="mySqlDataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://127.0.0.1:3306/customeradmin"/>
        <property name="username" value="root"/>
        <property name="password" value="mySQL" />
</bean>

<bean id="customerDao" class="com.infosupport.springfeatures.dao.impl.ibatis.CustomerIbatisDaoImpl">
        <property name="ibatisTemplate" ref="sqlMapClientTemplate"/>
</bean>

<bean id="orderDao" class="com.infosupport.springfeatures.dao.impl.ibatis.OrderIbatisDaoImpl">
        <property name="ibatisTemplate" ref="sqlMapClientTemplate"/>
</bean>

<bean id="sqlMapClient"  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation"
                value="classpath:com/infosupport/springfeatures/dao/impl/ibatis/sql-map-config.xml" />
        <property name="dataSource" ref="mySqlDataSource"/>
</bean>

<bean id="sqlMapClientTemplate"  class="org.springframework.orm.ibatis.SqlMapClientTemplate">
        <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

Code fragment 4: Configuring IBatis DAOs

With the above discussed participants we can easily reconfigure the container by using the newly created persistency-ibatis.xml instead of the old persistency-hibernate.xml file. The container automatically injects the IBatis DAOs in the business POJOs and the application continues to execute. Note that the common "*data access exception*" participant also prevents any ripple effects of the change to the business layer.

Analysis "*Data access exceptions*" participant	
Quality attribute Analysis	The common data access exception hierarchy of Spring supports modifiability, usability and testability. Modifiability because no ripple effect of changes to the business services occurs when replacing the persistence technology. It furthermore supports usability because exceptions thrown do not need to be catched. Exceptions thrown by a persistency framework are often checked, which means they have to be catched. However, data access exceptions are often unrecoverable and only lead to extraneous catch or throw clauses that clutter up your code. The latter also having a negative impact on testability with respect to unit testing, because the clutter up code needs to be tested as well. This is probably only valid if the exception can actually be handled. Unit testing classes is thereby enhanced.
Negative side effects	This flexibility is also a danger regarding usability, because developer must take their own responsibility of wondering if they actually need to catch this exception and how to recover. When exceptions are checked, developers are forced to think about handling exceptions in an appropriate way. Another side-effect is that if logging is a requirement it should occur before the exception is thrown and one does not always have access to that code.
Trade-off analysis	The temptation to not catch an exception and move on with development is enlarged by the "*Data access exception*" participant. However, developers are also tempted when one is obligated to catch a checked exception. In the latter case developers, for instance, simply place a comment line in the catch clause and done. The present author believes that each developer can handle this responsibility.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

Analysis "*SqlMapClientTemplate*" participant	
Quality attribute analysis	Spring supports a Blackbox and Whitebox approach towards using its IBatisSupport. See §2.2 for a discussion on which one to use. Both abstract away from the internals of the IBatis APIs, which increases the usability of this persistency mechanism. Modifiability is also enhanced because the support for IBatis uses the general Spring data access exceptions (see Table 6) and the general transaction management of Spring. Therefore, when the persistency layer is replaced by another, there is no impact on the business services. Both exceptions and transactions not need to be replaced by technology specific ones. Of course by injecting the

	dependencies, mocking is also enhanced.
Negative side effects	The common scenarios are supported by the Spring abstractions. However, if more sophisticated features need to be implemented, such as batch processing, the APIs are not sufficient [JHA05]. Then a combination of IBatis and JDBC can be used. The abstraction by Spring creates an indirection which has a negative effect on performance.
Trade-off analysis	The general support of transactions and data access exceptions and the usability of the Spring IBatisSupport are benefits one should exploit. If advanced features are needed then Spring can be configured to use a combination of JDBC and IBatis at the cost of an increase in complexity. With respect to the downgrade in performance, we argue that this is insignificant. The indirection is only needed to load the IBatis classes into the framework, which is accomplished during start up of the container.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

Analysis "*Separate configuration files*" participant	
Quality attribute analysis	Splitting up the application context over different XML files enhances modifiability, testability and usability. All beans in one file could lead to an unwieldy large file that is not maintainable. By separating the configuration files, scenarios as the above can also be easier implemented, because one can replace an entire XML file that contains the Hibernate DAO configuration, by an XML file that contains the IBatis DAO configuration. The latter is also useful in unit testing. For instance, when mocking the persistency layer.
Negative side effects	Applying this participant implies the need for a form of logging, because if one of the XML configuration files contains a fault, the container will throw an exception: "*could not start-up by a fatal exception*". When logging is applied more information will be available. Another shortcoming is that the container cannot check for the existence of bean definitions in another XML file.
Trade-off analysis	Using some form of logging is considered best-practice in many applications. The first disadvantage is therefore negligible. The second disadvantage is only when the initialization of a certain bean relies on the existence of another bean in another file and it should be checked during start-up of the container. Consider that beans are created by the container on their first request.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

Analysis "*Autowiring*" participant	
Quality attribute analysis	Autowiring increases usability, modifiability and testability. Usability is increased because developers do not need to explicitly type in all the dependencies which can be a cumbersome task. It increases modifiability and testability, because entire XML files can be replaced by mocking XML files for instance without having to adapt the existing XML file configurations.
Negative side effects	Autowiring also affects modifiability in a negative way, because of the loss of transparency. Removing explicit declarations also means removing a form of documentation, thereby introducing potential problems during modifications. Secondly, autowiring also has an impact on the performance considering it heavily relies on reflection.
Trade-off analysis	Using auto-wiring for purposes of the scenario described above (that is wiring between different layers) certainly provide benefits with respect to modifiability and testability. However, in the case of large bean definition files that describe the domain autowiring is not advised because it could become very complex and not maintainable. Another often propagated reason to use autowiring by default is that developers are not obligated to the cumbersome task of typing the dependencies. However, lots of tools exist today that provide the developers with the means of auto-completion for XML files. We generally recommend applying auto-wiring with care.
QA summary	Availability 0 Modifiability -+ Performance - Testability + Usability -+

6.3.3 MO-1.4 Adding logging functionality to multiple business POJOs

The reasoning framework only has two stateless businesses POJOs: "*CustomerServiceImpl*" and "*OrderServiceImpl*". As discussed in §6.3.1, Spring provides a "*ProxyFactoryBean*" to create proxies and apply advice to a class. This works fine for small applications since there are not many classes that need to be advised. Remember that for each bean definition one needs to explicitly configure which advices must be applied (see Code fragment 1). When the number of existing POJOs grows it becomes a more cumbersome task. Therefore, Spring provides other ways to configure the POJOs to be advised.

There are two ways, of which we will discuss the most powerful, namely the "*DefaultAdvisorAutoProxyCreator*". Code fragment 5 shows how this participant is configured. After the bean definitions are read in by the ApplicationContext, the "*DefaultAdvisorAutoProxyCreator*" searches the advisors in the context. It then applies the found advisors to any beans that match the advisor's pointcut. Note that the advisor specifies which advice should be applied and to which classes and methods within these classes. In our case all classes that end with "*ServiceImpl*" and all methods in these classes.

```
<bean id="loggingInterceptor" class="com.infosupport.springfeatures.advices.LogAdvice"/>

<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
   <property name="advice" ref="loggingInterceptor"/>
   <property name="pattern" value=".+Service\..+"/>
</bean>

<bean id="autoProxyCreator" class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"
</bean>
```

Code fragment 5: DefaultAdvisorAutoProxyCreator

Analysis "*Autoweaving*" participant	
Quality attribute analysis	Autoweaving mainly addresses usability and modifiability. Usability is enhanced because developers do not have to write advice configuration for each bean definition. The increase of modifiability is that when developers have written their POJOs these automatically benefit from crosscutting code.
Negative side effects	The danger also regards modifiability. By using Autoproxying, developers give up full control and loose some documentation. During maintenance or the addition of new classes, it could happen these are advised leading to unwanted behaviour. Another drawback is the impact on performance, considering that this feature also relies heavily on reflection.
Trade-off analysis	Autoproxying, just as autowiring, should be applied with great care. Wit respect to performance the impact on starting up the container is considerable. However, this should normally be a one time action and therefore not a reason for not using this participant.
QA summary	Availability 0 Modifiability -+ Performance - Testability 0 Usability +

6.3.4 MO-3.1 Exposing CustomerService as web service

Spring relies on another framework named XFire[11] for exposing a service as a webservice. XFire dynamically at runtime exposes a web service according to the information in the interface of the class that needs to provide its service and some extra information that can be configured. Code fragment 6 shows how a bean is configured as a web service with the use of Spring and XFire.

```
<bean id="customerService" class="com.infosupport.springfeatures.service.impl.CustomerServiceImpl"/>

<bean id="customerServiceExporter" class="org.codehaus.xfire.spring.remoting.XFireExporter">
        <property name="serviceFactory" ref="xfire.serviceFactory"/>
        <property name="xfire" ref="xfire"/>
        <property name="serviceBean" ref="customerService"/>
        <property name="serviceInterface" value="com.infosupport.springfeatures.service.ICustomerService"/>
</bean>
```

Code fragment 6: Exporting CustomerService as web service with XFire

[11] http://xfire.codehaus.org/

Based on the interface and the additional configuration information, Xfire in cooperation with the Spring "*ProxyFactoryBean*" (§6.3.1), creates a proxy that stands before "*CustomerServiceImpl*" The proxy then handles all technical concerns as retrieving and sending data. The actual remote requests are subsequently delegated by the proxy to the "*CustomerServiceImpl*" class. Note that the "*CustomerServiceImpl*" class is not aware that it's being exposed as a webservice; that is, there is no code in "*CustomerServiceImpl*" which is infrastructural related.

Analysis "*XFire Exporter*" participant	
Quality attribute analysis	Exposing a class as a webservice, only through configuration, separates the technical concerns from the functional ones and thus increases modifiability. It also enhances modifiability, because no code is generated that can be (accidentally) adjusted by someone else. It also increases testability, because during unit testing developers are not required to start an actual container to be able to test the functionality. Naturally integration testing does require the container.
Negative side effects	Performance and memory consumption is addressed negatively, because the proxy needs to be generated and kept in memory.
Trade-off analysis	The impact of this participant on performance is minimized. During start up of the container Spring creates the proxy and then all calls occur through it. This impact is also negligible considering that the overhead of using web services in the first place is significant.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

6.3.5 MO-3.2 Communicating with existing payment webservice

Spring provides a "*JaxRpcPortProxyFactoryBean*" that can be used to configure the settings needed to communicate with the external web service. Code fragment 9 shows the configuration settings for the current scenario.

Subsequently, the "*orderWebservice*" bean can be injected in the class that depends on it. The latter is unaware of the fact that "*orderWebservice*" is a webservice. Traditionally however, the client of the web service is forced to catch any remote exceptions. The Spring framework provides developers with an own "*RemoteAccessException*" participant which is unchecked (does not require catching the exceptions). The fragment below therefore contains a serviceInterface that points to the local interface and the portInterface property that uses the actual remote interface. The difference between these two is that the local interface does not specify that each method throws a "*java.rmi.RemoteException*". Spring subsequently catches the exceptions thrown by the Remote interface and converts these to its own unchecked exceptions, leaving it entirely up to the developer to catch those exceptions are not.

```
<bean name="orderWebservice" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
  <property name="serviceInterface" value=" com.infosupport.springfeatures.localInterfaces.LocalOrderFacade " />
  <property name="portInterface" value="com.buyabike.ejb.bus.RemoteOrderFacade"/>
  <property name="wsdlDocumentUrl" value="http://student719:9080/services/OrderFacade/wsdl/~
                                                                    ~OrderFacade.wsdl"/>
  <property name="namespaceUri" value="http://bus.ejb.buyabike.com"/>
  <property name="serviceName" value="OrderFacadeService"/>
  <property name="portName" value="OrderFacade"/>
</bean>
```

Code fragment 7: Exporting CustomerService as web service with Xfire

Analysis "*JaxRpcPortProxyFactoryBean*" participant	
Quality attribute analysis	The "*JaxRpcPortProxyFactoryBean*" is devised from "*ProxyFactoryBean*" and therefore at least exhibits the qualities discussed in §6.3.1. In addition, this participant enables us to inject a webservice into a bean, without that bean being aware of the fact that it is receiving a webservice. Thereby increasing modifiability and testability, because one can easily switch between objects that communicate remotely and ones that invoke local methods. The "*RemoteAccessException*" participant addresses modifiability, usability and testability in the same way as discussed in the analysis of the "*Data access exception*" participant (§6.3.2).
Negative side effects	As described in the analysis of the "*Data access exception*" throwing unchecked exceptions requires developers to make explicit choices regarding to catch exceptions or not. It could occur that developers do not catch exceptions that could efficiently be handled, also see (§6.3.2: Analysis data access exception).
Trade-off	The temptation to not catch an exception and move on with development is enlarged by the

analysis	"*RemoteAccessException*". However, this temptation is also there when one is obligated to catch a checked exception. In the letter case developers, for instance, simply place a comment line in the catch clause and done. The present author believes that each developer can handle this responsibility.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

6.3.6 MO-3.10 Communicating with a stateless session bean

As shown in §6.2, the IBM RAD 5.1 application server contains several remote stateless session beans, among which the "*OrderFacade*" session bean. Methods on this session bean should be remotely accessible for "*OrderServiceImpl*". The Spring framework therefore provides the "SimpleRemoteStateless-SessionProxyFactoryBean" participant. This participant can be used to encapsulate a remote stateless session bean. Code fragment 8 shows how it needs to be configured.

```
<bean id="orderFacadeBusiness" class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxy~
                                                                                    ~FactoryBean">
 <property name="jndiName" value="cell/nodes/localhost/servers/server1/ejb/com/buyabike/ejb/bus/~
                                                                                    ~OrderFacadeHome"/>
 <property name="jndiEnvironment">

  <props>
   <prop key="java.naming.factory.initial" value="com.sun.jndi.cosnaming.CNCtxFactory"/>
   <prop key="java.naming.provider.url" value="corbaloc:iiop:student719:2809"/>
  </props

 </property>

<property name="businessInterface" value= "com.infosupport.springfeatures.localInterfaces.LocalOrderFacade"/>
</bean>
```

Code fragment 8: Encapsulating stateless session bean

The jndiEnvironment property is needed to specify the location of the application server that exposes the stateless session beans. The jndiName property defines where the stateless session bean resides inside JNDI. The "*SimpleRemoteStatelessSessionProxyFactoryBean*" instance is configured with the JNDI location of the EJB remote home, as well as the business interface, "*LocalOrderFacade*". The proxy factory bean produces as its output a proxy object that implements the configured business interface. In a default configuration, at the time the proxy is created, the EJB home is looked up and cached. When a client of "*LocalOrderFacade*" invokes a business interface method on the proxy, the latter creates an instance of the session bean by calling create() on the cached home. It then invokes the business method on the session bean and returns the result, making sure the session bean instance is destroyed first.

Just as described in §6.3.5, this participant is also able to convert EJB specific checked exceptions into Spring unchecked exceptions. For a description of this mechanism see §6.3.2 and §6.3.5.

Analysis "*SimpleRemoteStatelessSessionProxyFactoryBean*" participant	
Quality attribute analysis	Spring provides a mechanism to integrate with existing stateless session beans on which many software systems rely. The implementation of the Spring bean that uses this stateless session bean is unaware of the fact that it communicates remotely. Thereby making it easy to temporarily switch the stateless bean for another instance (local or remote). This addresses both modifiability and testability.
Negative side effects	As agreed on by the analysis of a number of participants, it impacts performance because of the indirection.
Trade-off analysis	The performance impact is minimal. Spring provides a fairly simple service abstraction layer that makes it possible to configure the use of stateless session beans and inject them as if it were POJOs.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability 0

6.3.7 TB-2.2: Testing persistent data without modifying the persistency state

Unit tests for data access methods that access a real database have the problem that the persistence store's state is affected. All kinds of tools, such as dbUnit[12], help in temporarily replacing the real database by a development database. However, then still changes to the database state, made in unit tests, affect the future state of this persistence store. Spring provides support through the "*AbstractTransactionalDataSourceSpringContextTests"* participant. Spring developers need to subclass their unit test class from this class. By default each unit test method in the class is then executed in a transaction. Just before the test method ends, the transaction will be rolled back, thereby not modifying the persistence store.

The "*AbstractTransactionalDataSourceSpringContextTests*" requires the existence of a configured "*datasource*" and a "*TransactionManager".* Spring relies on both to have each unit test executed in a transaction.

Analysis "*AbstractTransactionalDataSourceSpringContextTests*" participant	
Quality attribute analysis	Developing unit tests for data access methods is simplified, thereby enhancing usability. Testability is also increased, because unit test methods only contain the actual test and not other code needed to revert to the old persistence store state.
Negative side effects	The unit test class becomes dependent on the existence of the container, thereby decreasing modifiability to another container. Secondly, this participant is developed to only work with a single database. When adapting this unit test to work with multiple databases at the same time, one needs to create its own implementation.
Trade-off analysis	Using this participant greatly aids in developing data access test methods. When existing unit test need to be adapted for multiple databases then the code needs to be adapted. However, this implies that one has to develop its own mechanisms, what needed to be done without this participant anyway.
QA summary	Availability 0 Modifiability + Performance 0 Testability + Usability +

6.3.8 MO-3.7 Integrating Spring with Eclipse Rich Client

An existing Eclipse Rich Client needs to communicate with the "*OrderServiceImpl*" to acquire a list of orders. This list is subsequently shown in one of the forms of the client. The communication protocol used is "*HTPP Invoker*".

The first step in accomplishing this scenario is to export "*OrderServiceImpl*" as a "*HTTP Invoker*" service. Spring provides the "*HTTPInvokerServiceExporter*" participant, that is responsible for creating a proxy on which clients can invoke methods. Code fragment 9 shows the configuration of this participant.

```
<bean id="orderServiceHTTPInvoker" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <property name="service" ref="orderService"/>
        <property name="serviceInterface" value="com.infosupport.springfeatures.service.IOrderService" />
</bean>
```

Code fragment 9: Exporting OrderServiceImpl as HTTP Invoker service

Despite the fact that the above configuration required setting less properties, it embodies the same principles as exposing a webservice with Xfire (see §6.3.4). The service property is a reference to the bean that needs to be exposed and the serviceInterface property defines the interface that the referenced bean should implement. Based on the configured items a proxy is created that handles all remoting issues and delegates to the actual bean for business functionality. This scenario is also implemented with use of RMI, but not further discussed. One aspect of interest to notice is that when the container starts up, the OrderService is exported with RMI and HTTP Invoker through two different proxies, without OrderService knowing about one of them.

The Eclipse Rich Client also needs to be adjusted in order to communicate over "*HTTP Invoker*" with the OrderService. Therefore a "*HTTPInvokerProxyFactoryBean"* participant must be configured as shown in Code fragment 10.

[12] http://dbunit.sourceforge.net/

```
<bean id="orderService" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <property name="serviceUrl" value="http://localhost:8080/SpringFeatures/OrderServiceHTTPInvoker.htm"/>
        <property name="serviceInterface" value="com.infosupport.springfeatures.service.IOrderService"/>
</bean>
```

Code fragment 10: Accessing a HTTP Invoker based service

Up till now the container loaded an ApplicationContext. The container subsequently injected dependencies in the beans according to the configuration files. The Eclipse Rich Client project however is not web based and doesn't have an ApplicationContext. But the Spring framework can also be used in J2SE like environments. The following code shows how Spring is used to request for an orderService bean in the actual code. As can be seen, the Spring container can be loaded with a bean configuration file and one can subsequently request the container for bean instances. Note that this is the first time that the business code is aware of the framework.

```
public ArrayList<OrderDTO> getAllOrders()
{
        BeanFactory ctx = new ClassPathXmlApplicationContext("ctx.xml");
        com.infosupport.springfeatures.service.IOrderService orderService =
                (com.infosupport.springfeatures.service.IOrderService)ctx.getBean("orderService");

        List<Order> orders = orderService.getAllOrders();
        ....
        ...
```

Code fragment 11: Accessing a HTTP Invoker based service

Considering that "*HTTPInvokerServiceExporter*" and "*HTTPInvokerProxyFactoryBean*" are implemented in the same way as respectively "*Xfire Exporter*" (§6.3.4) and "*JAXRPCProxyFactoryBean*" (§6.3.5), we refer to them. The lesson learned in executing this scenario is that the Spring hooks are used in a consistent fashion. Considering that HTTP Invoker is a protocol developed by Spring and not an existing remoting framework, this one will be analyzed.

Analysis "*HTTP Invoker*" participant	
Quality attribute analysis	The "*HTTP Invoker*" protocol uses HTTP for transport and therefore does not conflict with firewalls. This enhances modifiability, because changes to network security do not affect the correct execution of services based on it. Modifiability is also increased because the protocol uses Java's standard object serialization. This has the benefit that if the arguments or return types are complex they can still be serialized as opposed to some communication mechanisms that rely on their own serialization mechanism.
Negative side effects	The protocol is not yet widely used. A condition for using this protocol is that the developer should have access to the client's code as well as the server's code. Secondly, it negatively addresses performance. Some remoting
Trade-off analysis	If the objects that need to be send over the wire are complex, then "*HTTP Invoker"* is a good choice if one has access to the client's as well as the server's code.
QA summary	Availability 0 Modifiability + Performance 0 Testability 0 Usability 0

6.3.9 Usability scenarios

For development of the framework-based application and the implementation of the different scenarios we used the Eclipse IDE[13]. With respect to US-1.1 (refactoring of class names) Eclipse provides all kinds of refactoring capabilities, but the one perhaps mostly used is the refactoring of a class name or a class attribute name. We expected that this was not possible anymore due to the extensive use of configuration files. However, if we renamed one of them then the bean definitions were also adjusted and the application would still be deployed successfully. This IDE feature is specific to Eclipse, it could be that other development environments do not provide such functionality. Another IDE feature is auto completion which is required by scenario US-1.3. Eclipse on itself does not provide auto completion functionality when defining

[13] http://www.eclipse.org/

beans. However, the open source community developed an eclipse plug-in named Spring IDE[14]. This integrates with Eclipse and thereby provides all kinds of extra features specific in helping developers define a bean definition. These conclusions are however only appropriate when using Eclipse. It is expected that the commercial IDE's do not provide these extra features. When these features are not provided then using Spring strongly decreases usability.

Regarding US-1.2 (the impact of the framework on debugging) we note that Spring does have an impact on debugging your application. As discussed in §2.3, frameworks are harder to debug in general. Spring mostly influences debugging by the extensive use of proxies. When stepping through the application, one often also needs to step through the proxy code which abstracts away from specific application details. Another observation is that one needs to step through the framework code when a container fails to start-up by a fault in the configuration files. At this time, there is no debugger that would permit stepping through an xml bean configuration file showing the necessary debugging information.

The Spring community is very large. There are numerous forums available of which the most notable is the Spring forum[15]. During development we encountered a number of difficulties which were addressed in that forum. It is also notable that the initiators of Spring, among Rod Johnson, is also still active in answering questions. With respect to US-1.4 we can conclude that it is met.

6.3.10 Summary results

The results achieved are summarized in the next sub sections for each quality attribute of our analysis. In addition the high-level goals provided in §6.1.1 are discussed throughout this section.

6.3.10.1 Availability

Availability was addressed by just one scenario. It is therefore very hard to conclude if high-level goal "*HG-7) provide effective and reliable operation at many different scales*" (§6.1.1) is met by the framework. With respect to the implementation of AV-1.1 we can conclude that failing-over, implemented with Spring, does not require deployment to an application server. The implementation of this scenario is therefore not tied to a specific application server which enhances modifiability. After all, the code can easily be ported without loosing any functionality. We should also note that application servers often provide clustering and load balancing functionality that address fail-over scenarios and load distribution. Spring applications, do not have to, but can be deployed to numerous application servers [JHA05]. What the Spring framework offers for deployment onto an application server and what the impact is for the software quality is future work.

6.3.10.2 Modifiability

Modifiability was addressed by five scenarios in total. Also, most other framework participants discussed in other quality attribute scenarios impacted modifiability in some way. The main observation was that the Spring framework participants realize a POJO programming model. This implies that the business domain classes remain simple POJOs that are unaware of the fact that they are exported, realized within transactions or other technical (infrastructural related) concerns. There are three general means by which Spring accomplishes this model and enhances modifiability.

At first, most participants use dynamic generation of proxies to add functionality to a class, for instance exporting the class as a webservice. Because of the fact that the exported class is unaware it's being exported, its functionality can be reused in other environments. On top of that, one can export the same class for a number of protocols without changing anything in the actual code, thereby fully reusing the code.

Secondly, Spring provides a number of own exception hierarchies, for instance, the "*Data access exceptions*" participant. These hierarchies benefit modifiability in two ways. At first, they are common for a number of underlying technologies (among Hibernate, JDBC and IBatis). If a persistency framework changes, then no ripple effects occur to the code that handles the throwed exceptions. Secondly, the checked exceptions thrown by the underlying framework are modified to unchecked exceptions by Spring. The primary benefit is that the calling class does not need to catch the exceptions, thereby allowing for instance, to substitute a remote service by another remote service or even a local one.

Thirdly, if there are still APIs that need to be used directly from the code (such as Hibernate, JavaMail, JDBC etc.), Spring provides its own participant abstractions that can be used in a black- or white box approach. The former being the one we prefer, because of the reasons provided in §2.2.

---

[14] http://www.springide.org/project
[15] http://forum.springframework.org/

The way in which the POJO programming model is accomplished ensures that applications built with Spring are easily portable between different environments. The only requirement is that the environment should support Java. Naturally, when using one of the service abstraction layers over APIs one is tied to the Spring container. Remember that one scenario also applied Spring dependency injection and the "*HTTP Invoker*" participant in a J2SE environment. This same scenario also verified that Spring can be used in combination with other frameworks, thereby addressing high-level goal "*HG-4) Foster integration with other frameworks*".

In addition, we also implemented a number of scenarios that were related to remote communication with other systems. Spring provides a number of protocols to achieve remoting and thereby also meets high-level goal *"HG-1) Provide support for remote communication interoperation. The framework should support various ways for remote communication interoperation across different vendor implementations."*

We conclude that high-level goal "*HG-2) increase modifiability of J2EE framework-based applications"* is met by the framework.

6.3.10.3 Performance

With respect to performance and the relating high-level goal "*HG-8) provide efficient and predictable execution*", we have to note that no scenarios that directly affect performance were executed. The primary reason was those received a low priority. Despite this, numerous framework participants have a slight but negative effect on performance. This is mostly due to the frameworks' heavy reliance on reflection for dependency injection and the runtime creation of proxies. However, the impact on performance is mostly limited to the creation of objects, which occur on their first request. Thereafter, the container reuses these created proxies and the impact can be considered negligible. To determine the exact impact, more measurements are needed.

6.3.10.4 Testability

One scenario primarily addressed testability with respect to unit testing. On top of that, multiple Spring participants used in other scenarios also positively affect this quality attribute. The reasons for this are manifold. At first, dependency injection ensures that dependencies of a class can be easily substituted by a mock implementation when one programs to an interface. Secondly, the extensive use of AOP separates the infrastructural related code, from the business code, and ensures that developers can focus their unit testing efforts on the business logic. In addition, by applying AOP one can unit test the functionality of the aspect in isolation. If a traditional approach was used, then that functionality needed to be tested for each point in the program it was executed, which can become a cumbersome task. Thirdly, Spring provides a number of participants that help in simplifying testing as discussed in the analysis of the participants "*Separate configuration files*" (§6.3.2) , "*Autowiring*" (§6.3.2) and "*AbstractTransactionalDataSourceSpringContext-Tests*" (§6.3.7). Finally, when using Spring the beans remain simple POJOs, which means that one does not need to startup the container for executing the unit tests. Starting up the container often requires much time, which discourages developers to continuously run their unit tests. In conclusion we believe that Spring contributes in developing J2EE applications that are better testable, which contributes to high-level goal "*HG-3) facilitate in testing activities*".

6.3.10.5 Usability

A number of participants discussed in the scenarios affected usability in a positive way. For instance, we think that not obligating developers to catch exceptions they cannot handle or will not handle anyway is a good thing. As agreed on before, this does require developers to acknowledge their own responsibility. Spring also provided "*autowiring*" and "*autoweaving*" that help in configuring the bean definition files. The scenarios that primarily addressed this attribute were positively executed also, which is primarily because Spring is supported by a large community. This same community is also responsible for the development of tooling support specific for developing applications on top of the framework. However, we should conclude that the tooling support is, by our knowledge, limited to the Eclipse IDE. In general we believe that Spring supports high-level goal "HG-1) *Simplify and increase productivity in developing J2EE applications*" because of the many participants it provides that can be reused and because using the framework minimizes impact on development. In addition, the consistent way in which Spring provides its participants (mostly through dependency injection, Spring AOP and the service abstraction layers) and the large support of the community (documentation, articles and forums) it also meets high-level goal "*HG-9) Minimize learning curve*".

Chapter 7

Analysis FSQAM

This chapter evaluates the application and results of FSQAM to the Spring framework. Each phase, presented in Figure 6 will be discussed in section 7.1 up to section 7.3.

7.1 Collect concrete scenarios

7.1.1 Phase 1: Identifying context and high-level goals

The goals explicitly mentioned in the Spring's documentation were extracted and validated by the ones identified in Chapter 2 and Chapter 3. The outcome was that HG-2, HG-5, HG-6, HG-7 and HG-8 could be added to the list. The question that rises is why Spring does not mention these goals in their documentation or why we could not identify them. The main reason is that Spring mention's these implicitly throughout its documentation. Most of the high-level goals explicitly stated are based on the setbacks in traditional J2EE development. For instance, the framework should facilitate in testing and debugging activities (HG-3). However, this does not imply that Spring does not aim for providing remoting support for interoperation (HG-6). We also noticed that if we would not have had the goals of frameworks (§2.3), middleware infrastructures (§2.5) and J2EE (§3.2), the outcome of this first phase would not have been complete and probably needed to be adjusted during the execution of subsequent phases. The latter requiring unnecessary feedback to the stakeholders or re-conducting an interview because scenarios related to this new goal need to be gathered.

Most of the quality attributes defined are typical quality attributes in an analysis. However, we have shown that usability, as defined in this research, is of typical use in evaluating frameworks. For instance, the extensive use of XML by a framework also implies the need for additional tooling. Without this extra tooling, developers are limited in the functionality provided by IDEs, thereby negatively impacting high-level goal HG-1: "*Simplify and increase productivity in developing J2EE applications*".

7.1.2 Phase 2: Identifying the architectural building blocks

We experienced that this phase could be never-ending. A framework, such as Spring, encompasses an enormous amount of interesting functionality. During studying the first Spring book, one could simply continue reading into all kinds of details. Determining when to continue to the next activity is difficult and has an impact on the outcome of the interviews. The main reason is that the more knowledge is gathered regarding the possibilities of the framework, the more the evaluator can direct interviews to find scenarios of interest. That is, scenarios that are supported in some way by the framework. A benefit of having this much knowledge is that the interviewing time can be used optimally. Another benefit is that the framework also encompasses knowledge with regard to J2EE development, that can be used. For instance, we identified a scenario (MO-3.10) in which HTTP Invoker is used as the remoting protocol. This protocol is Spring specific and the scenario was added after discussing it with some stakeholders. The drawback is that a lot of knowledge gathered during this activity will perhaps not contribute to the analysis. We propagate that this phase should only result in a list of architectural building blocks of the framework. The more fine-grained participants need to be identified during the "mapping" activity, thereby optimizing the precious analysis time. Additional scenarios that contain Spring specific participants could then still be added.

7.1.3 Phase 3: Prototyping

When looking at the gathered scenarios it is notable that most scenarios primarily affect modifiability. This is probably due to the decision to keep the prototype as small as possible. This offered most stakeholders a trigger to come up with modifiability scenarios. An example of such a scenario was that the business and data tier needed to be deployed to an application server in order to scale the application with clusters. Note that this scenario was also a trigger for the interviewer to ask why an application should be clustered, thereby identifying new (availability) scenarios. The views we primarily used, logical, physical and the context view (§4.3.2) also lend themselves mostly for collecting modifiability scenarios. The latter is also responsible for the large amount of interoperation scenarios. Initial interviews with our key stakeholder

indicated that performance would not be of great concern within this research. We therefore did not use a process view that explains the minimal amount of performance scenarios addressed.

7.1.4 Phase 4: Interviewing

Subsequently, we collected a set of scenarios by conducting interviews with different employees at Info Support. As stated in §6.1.4, we did not have the resources available to carry out an interview with stakeholders that represent different groups, such as maintainers, testers etc. This setback was addressed by verifying if all of the high-level goals are addressed by at least one scenario. For instance, after the first two interviews it appeared we did not have any scenarios related to integrating two frameworks (HG-4). Subsequently, we could direct stakeholders by asking questions that were related to the goals not yet addressed.

We do not state that the found scenarios (listed in the Fourth Addendum) are complete. As described in the previous section we would probably have found more by using other or additional views. However, the ones we have found are validated by the high-level goals and our key stakeholder. We could have continued with extraction of scenarios, but then not enough time would have been left for the subsequent phases to achieve results of practical value to Info Support. Just as in identifying the architectural building blocks, it is also hard to determine when to end this activity. The decision to stop with interviewing was determined in cooperation with our key stakeholder.

7.1.5 Phase 5: Prioritizing

During the next phase, prioritizing, we narrowed the analysis scope from 36 scenarios back to eight. The first thing to notice is that the selected scenarios do not widely differ (from each other), which is due to the fact that only one stakeholder was involved in prioritizing. In other situations it would be best to have all stakeholders involved in prioritizing the scenarios, which would probably lead to a selection of scenarios that are more different and address more different concerns. However, this is not the result of our proposed approach but due to the setup in which this case study is executed.

When narrowing done the scope to eight scenarios it looks impossible to make statements about the software quality of this framework. We, however, believe this is not completely true. At first, determining the ramification on quality attributes of the framework participants belonging to these scenarios are also considered the most important ones. Secondly, these scenarios were to such a degree large that the most reoccurring framework participants were assessed. This is validated by the fact that the participants of this phase reoccur in the scenarios currently not analyzed. We should note that this statement is based on an analysis not as comprehensive executed as the ones in the mapping activity. It is actually more based on current experiences with the framework.

This observation leads us to the discussion of finding the optimal number of scenarios; the scenarios which together address all the framework participants of interest which are used in accomplishing all the extracted scenarios. However, we believe this is impossible because of the limited time required in determining what those participants are, which is exactly addressed by prioritizing.

Prioritizing by the evaluator leads to finding the most important scenarios. We also prioritized according to the time we believed it would take to implement a scenario. The scenarios that are the most important and difficult to realize should receive the most attention. If not all of these can be evaluated according to the planning, then less difficult scenarios can be analyzed for making most of the time.

7.1.6 Phase 6: Mapping framework participants to scenarios

During this phase we did not find any scenarios, in the selected set, that were not supported by the framework. This is due to the large popularity of – and support for the framework. In time, lots of functionality is added to the Spring framework. Another reason is that the selected set of scenarios by coincidence is supported. Nevertheless, when evaluating other J2EE meta-frameworks this categorization could prove its use.

The role of the general tactics was limited to function as a trigger. For instance, the “prevent ripple effects” tactic states that when making a modification to module *A*, one should not need to make changes to modules which depend on it [BCK03]. This tactic was interpreted by us as the use of the dependency injection participant, which propagates that properties are interfaces that need to be set by an implementation of that interface. When defining our approach we had expected to benefit more from these tactics. For instance, in [LBK01], the authors successfully mapped EJB participants to general tactics. However, most of these general tactics, were applicable to functionality of EJB in combination with an application server. For instance, the application server was responsible for load balancing, fail-over scenarios through the “heartbeat” tactic. Application server like functionality was beyond the set of our

selected scenarios.

7.2 Specify a framework-based application

The views developed in this phase were mostly based on the outcome of the prototyping activity. At first, we wanted to use these views in order to better determine the impact of the framework. However, during the development of the views we discovered that this was more difficult than we expected. To be able to do impact analysis on the views, the framework participants also needed to be mapped and described. This would have required the development of views that show the use of autowiring, autoweaving and a number of participants that are accessed by configuration. Considering the time available and the need for practical results we decided that the development of views, which shows the support of a framework, should be addressed in a separate research.

The views still have their function. They know support discussing the impact of each scenario onto the framework-based application. We also decided to implement the basic application and the scenarios for three reasons. At first, by implementing the scenarios we would validate if the mapping of framework participants was done right. Secondly, the source code delivered can be reused by other developers at Info Support for their own projects. Finally, actually implementing an application on top of the framework is the only way to determine the usability supported by the framework.

7.3 Analyze scenarios on framework-based application

The way in which we reported the results of our analysis for each scenario is twofold. At first, we documented how the implementation of a scenario is done with the framework. This documentation is valuable for other developers as a reference. In addition, we discussed the attribute ramification of using each participant together with rules of thumb in how and when to use them.

With respect to the remoting scenarios we identified that the framework provides the participants in an identical manner. One could argue that evaluating each of them is therefore unnecessary. However, identifying that the framework participants are offered in the same way is a valuable conclusion with respect to consistency that contributes to minimizing the learning curve. For readability reasons one could then refer to the software quality discussion of the other participant.

Another observation is that we discussed the use of "*HTTPInvoker*" at a different level then the other scenarios. This is mostly due to the fact that this protocol is Spring specific and how it is provided addresses the same qualities as for instance the "*JAXRPCProxyFactoryBean*" and was therefore not again discussed.

An additional aspect to take notice of is that no scenarios were discussed that primarily affect performance. This directly stems from our prioritizing activity in which no performance scenarios were selected. Despite, we made statements that the extensive use of reflection and the use of dynamic proxies have a negative impact on performance. We believe that exact measurements are necessary to determine if the performance poses a risk in real J2EE applications. However, this statement does not imply that the results are not valuable at all. At first, the output can be used for further analysis. For instance, we identified and implemented numerous scenarios that poses the risk of bad performance. These can be used as input for profiling. Secondly, the approach is used for qualitatively analysis of the software quality of which performance is just one aspect. Determining only the performance of a framework would be an entire separate activity.

The next aspect to discuss is if our results would have been different if other scenarios were analyzed, for instance more availability or performance scenarios. We believe that the choice of scenarios indeed has an impact on the end results. For instance, it could be that deployment to an application server ties the Spring container in a specific way to this server. This would decrease modifiability, because porting to a different environment would require changes. Therefore we believe that the results of FSQAM as applied to Spring are tied to the scenarios executed. This means that the results are only valid for the scenarios selected by our key stakeholder. If the results need to indicate the software quality of Spring in general, then more different scenarios need to be executed.

Chapter 8

Conclusion

8.1 Contributions

The research question addressed in this thesis is "*How to analyze the software quality of a J2EE meta-framework?*". The main contributions of our research are twofold: at first we define an approach for determining the software quality of J2EE meta-frameworks that is based on a literature study. Secondly this approach is applied to the Spring framework, which is of practical use for Info Support.

The application of FSQAM to the Spring framework indicated that the results of this method are twofold: at first, the software quality supported by the Spring framework is documented. Secondly, FSQAM resulted in an application (supported by documentation) that can be reused by other developers at Info Support as a reference. In addition to these primary contributions, applying FSQAM also resulted in a list of high-level goals, a list of concrete scenarios and a description of a framework-based application suited for software quality analysis. These intermediate products can be reused for analyzing other J2EE meta-frameworks.

The analysis of FSQAM to the case study also pointed out two aspects to consider. At first, the prioritizing phase ensures that the precious analysis time is spent the best. However, the results of the analysis could then be only tied to these scenarios and not to the framework as a whole. In order to determine the quality of the framework one needs to analyze as much diverse scenarios as possible. A second aspect to take notice of is that usability, as defined by us, is valuable in evaluating the software quality supported by a framework.

The primary observation with respect to the Spring framework was that modifiability, testability and usability were positively affected, at the cost of performance. With respect to performance we identified that most of the scenarios heavily rely on reflection and indirection therefore possibly negatively impacting performance. However, as concluded before, additional measurements onto our framework-based application should confirm this. Finally, it is also hard to conclude if availability is enhanced by the framework. We have discussed just one situation in which the framework offered a participant that helped in accomplishing a fail-over scenario. Again, future work should point out if availability is correctly addressed by the framework and at what cost.

The foundation of defining an approach as well as the application to Spring was based on an extensive literature study into frameworks in general, J2EE and software quality analysis methods. At the start of the project we and our key stakeholder had a number of questions that first needed to be answered. The contributions of this study are manifold. At first, we defined frameworks and identified how they can be used by developers and what their main benefits and issues are. In addition, we explained J2EE according to its reference architecture and showed a classification for J2EE frameworks. Next, we identified and discussed a number of software architectural analysis methods that is the basis for FSQAM. This literature study was not only useful for in acquiring the knowledge necessary for a successful completion, but it also foresees in the terminology used in the remainder of the document.

In conclusion we believe that FSQAM is successfully applied to the case study. The stakeholders, specially the key stakeholder, has indicated he is satisfied with the results. That is; not only the results regarding the case study, but also the results with respect to the definition of an approach and the conducted literature study.

8.2 Future work

We concluded about the software quality of the Spring framework with respect to the highly-prioritized scenarios by our key stakeholder. This ensures that we have given insight into the scenarios of importance for Info Support. However, in order to validate if our conclusions apply to the entire framework, we need to execute more scenarios that are availability and performance related. The framework-based application we have developed and the identified scenarios can be the foundation for this future work.

Future work could also include the application of FSQAM to other J2EE meta-frameworks. At first, others can then reuse our scenarios to determine if they provide a sufficient description of the non-functional requirements for J2EE applications. In addition, it would also be interesting to study if our approach is applicable for single-tier frameworks, such as web frameworks.

The analysis of the approach already indicated that it was hard to create architectural views that made it possible to determine the impact of framework participants. A separate research should study which views can be used to display framework related functionality that is meaningful enough for analysis. The presented results with respect to Spring can be used to validate the outcome of this future work. A benefit gained when analyzing the software quality based on views is a strong decrease in the time the evaluation will take. On the other hand, benefits such as having a reference application for developers and addressing usability of the framework will not be accomplished.

Bibliography

[BCK03] Len Bass, Paul Clements, Rick Kazman (2003) *Software Architecture in Practice Second Edition*, Addison-Wesley Professional, The SEI Series in Software Engineering, ISBN 0321154969, Pages 528

[BKB01] L. Bass, M. Klein, F. Bachmann (2001) *Quality attribute Design Primitives and the Attribute Driven Design Method*, Lecture Notes In Computer Science; Vol. 2290 Revised Papers from the 4th International Workshop on Software Product-Family Engineering Springer-Verlag, Pages 169 – 186

[DN02] L. Dobrica, E. Niemela (2002) *A Survey on Software Architecture Analysis Methods*, IEEE Transactions on Software Engineering, Volume 28, Issue 7 (July 2002), Pages 638-653

[FS97] M. Fayad, D. C. Schmidt (1997) *Object-Oriented application frameworks*, Communications of the ACM, Volume 40, Issue 10 (October 1997), Pages 32-38

[Hau97] J. Hautamaki (1997) *A survey of frameworks*, Department of computer science, University of Tampere, Pages total 36

[FHLS97] G. Froehlich, H. J. Hoover, L. Liu, P. Sorenson, *Hooking into Object-Oriented Application Frameworks*, Proceedings of the 19th international conference on Software Engineering (1997), ISBN 0-89791-914-9, Pages 491-501

[Fow04] M. Fowler (2004), *Inversion of Control Containers and the Dependency Injection pattern*, http://www.martinfowler.com/articles/injection.html

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994) *Design Patterns,* Addison-Wesley, ISBN 0201633612, Pages 395

[HF98] D. S. Hamu, M. E. Fayad (1998) *Achieving bottom-line improvements with enterprise frameworks,* Communications of the ACM, Volume 41, Issue 8 (August 1998), Pages 110-113,

[IEEE90] IEEE (1990) *Standard Glossary of Software Engineering Terminology* Std. 610.12-1990,

[J2EE03] Sun Microsystems (2003) *Java™ 2 Platform Enterprise Edition Specification v1.4*, Pages 246

[JF88] R. E. Johnson, B. Foote (1988) *Designing reusable classes*, Journal of Object-oriented programming, SIGS, 1, 5 (June/July. 1988), Pages 22-35,

[JH04] R. Johnson, J. Hoeller (2004) *Expert One-on-One J2EE development without EJB*, Wiley Publishing, ISBN 0764558315, Pages 578

[JHA05] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu (2005) *Professional Java Development with the Spring framework*, Wiley Publishing, ISBN 9780764574832, Pages 649

[JHA06] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet (2006) *Spring the reference documentation*, version 1.2.8, Pages 270,

[Joh05a] R. Johnson (2005) *J2EE development frameworks*, IEEE Computer, Volume 38, Number 1 (January 2005), Pages 107-110,

[ISE06] Info Support Endeavour (2006) *Use Case Rapport Buy-A-Bike*, Info Support B.V, version 1.1, Pages 13

[KBA94] R. Kazman, L. Bass, G. Abowd, M. Webb (1994) *SAAM: A Method for Analyzing the Properties of Software Architectures*, Proceedings of the 16th international conference on Software engineering, Pages 81-90,

[KKB98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere (1998) *The Architecture Tradeoff Analysis Method*, Proceedings of the fourth international conference on Engineering Complex Computer Systems ICECCS (August 1998),

[Kru95] P. Kruchten (1995) *Architectural Blueprints – The "4+1" View Model of Software Architecture*, IEEE Software, vol. 12, issue 6 (November 1995), Pages 42-50

[Lar02] Craig Larman (2002) *Applying UML and Patterns, An introduction to Object-Oriented Analysis and Design and the Unified Process Second Edition*, Prentice Hall PTR, ISBN 0130925691, Pages 627

[Lau02] Soren Lauesen (2002) *Software Requirements Styles and Techniques,* Addison-Wesley, ISBN 0201745702, Pages 591

[LBK01] A. Liu, L. Bass, M. Klein (2001) *Understanding the qualities of EJB,* Papers of DOA 2001 International Symposium

[Lee02] B. Leech (2002), *Asking questions: Techniques for semi structured interviews*, Political Science and Politics, vol. 35, no. 4, Pages 665-668

[LRV01] N. Lassing, D. Rijsenbrij, H. van Vliet, *Viewpoints on modifiability,* Division of Mathematics and Computer Science, Faculty of Sciences, Vrije Universiteit, Pages total 22

[MBF99] M. Matsson, J. Bosch, M. E. Fayad (1999) *Framework integration Problems, Causes, Solutions*, Communications of the ACM, Volume 42, Issue 10 (October 1999), Pages 80-87

[McC04] S. McConnell, *Code Complete second edition,* Microsoft Press, ISBN 0735619670, Pages 914

[MFB02] H. Mili, M. Fayad, D. Brugali, D. Hamu, D. Dori (2002) *Enterprise Frameworks: issues and research directions*, Software—Practice & Experience Volume 32 , Issue 8 (July 2002) , Pages 801-831

[Mil06] D. Mills (2006) *The Rise (and Rise) of the Meta-Frameworks and other Web Development trends*, http://www.bloggingaboutoracle.org/wpcontent/upload/Guru4ProMetaFrameworks.pdf

[Paw01] Monica Pawlan (2001) *Java 2 Enterprise Edition Technology Center*, http://java.sun.com/developer/technicalArticles/J2EE/Intro/

[WB05] Craig Walls, Ryan Breidenbach (2005) *Spring in Action*, Manning Publications Co, ISBN 1932394354, Pages 444

[Zar04] A. Zarras (2004) *A comparison framework for Middleware Infrastructures,* Journal of Object Technology, Volume 3, Issue 5, (May 2004), Pages 103-123

[ZBJ04] L. Zhu, M. A. Babar, R. Jeffery (2004) *Mining Patterns to Support Software Architecture Evaluation*, Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), Pages total 10

First Addendum

J2EE participant elaboration

Table 3: Summary of J2EE technology Components and Services [BCK03; J2EE03]	
Component/Service	Description
Java Naming and Directory Interface(JNDI)	J2EE's directory service allows Java client and Web tier Servlets to retrieve references to user-defined objects such as EJBs and environment entries as data sources.
Java Transaction API (JTA)	Makes it possible for EJBs and their clients to participate in transactions; updates can be made to a number of beans in an application, and JTA makes sure all changes commit or abort at the end of the transaction; relies on JDBC-2 drivers for support of the XA protocol and hence the ability to perform distributed transactions with one or more resource managers.
JavaMail	The JavaMail API provides a platform-independent and protocol-independent framework to build mail and messaging applications
Remote Method Invocation / Internet Inter-ORB Protocol (RMI/IIOP)	Provides developers with an implementation of Java RMI API over the OMG's industry standard Internet-ORB Protocol (IIOP); developers can write remote interfaces between clients and servers and implement them using Java technology and the Java RMI APIs.
Java Database Connectivity (JDBC)	Provides programmers with a uniform interface to a wide range of relational databases and provides a common base on which higher-level tools and interfaces can be built.
Java Messaging Service (JMS)	Provides J2EE applications with support for asynchronous messaging using either point-to-point or publish-subscribe styles of interaction; messages can be configured to have various qualities of service associated with them, ranging from the best effort to transactional.
Java Management Extensions (JMX)	JMX is provided for managing and monitoring a deployed application.
Java Webservices	J2EE provides full support for both clients of web services as well as web service endpoints. Several Java technologies work together to provide support for web services.

Second Addendum

Quality attributes definitions

This Addendum provides an explanation of the quality attributes addressed in our research: availability, modifiability, performance, testability and usability. In addition it provides the scenario generation tables identified in [BCK03]. Except for usability, considering that this definition differs from the one used in [BCK03].

Availability
The availability of a system is the probability that it will be operational when it is needed. Scheduled downtimes are not considered when calculating the downtime, since it is not needed by definition [BCK03].

Availability is among others concerned with: "how is system failure detected? How frequently is system failure detected? What happens when system failure is detected? How long is a system allowed to be out of operation? When may failures occur safely? How can failures be prevented? What kinds of notifications are required when a failure occurs?" [BCK03] The possible values for the availability general scenario are provided in table 1.

Table 1: Possible input for each component in an availability general scenario [BCK03]

Component	Possible values
Source	Sources of stimulus comes from: - Internal of the system - External to the system
Stimulus	A fault of one the following types occurs: - Omission; a component fails to respond to an input. - Crash; the component repeatedly suffers omission faults. - Timing; a component responds but the response is early or late. - Response; a component responds with an incorrect value.
Artifact	One of the following resources that need to be available: - System's processors - Communication channels - Persistent storage - Processes
Environment	The state of the system can affect the response: - Normal operation - Degraded mode (i.e., fewer features, a fall back solution)
Response	System should detect event and do one or more of the following: - Record it. - Notify appropriate parties, including the user and other systems. - Disable sources of events that cause faults or failure according to the defined rules. - Be unavailable for a pre-specified interval, where interval depends on criticality of system. - Continue to operate in normal or degraded mode.
Response Measure	Time interval when the system must be available. - Availability time. - Time interval in which system can be in degraded mode. - Repair time.

Modifiability
Modifiability` is about the cost of change. This attribute has two concerns: 1) what artifact is changed 2) when is the change made and who makes it?

Table 2: Possible input for each component in a modifiability general scenario [BCK03]	
Component	Possible values
Source	Modification request is done by: - End user - Developer - System administrator
Stimulus	Reasons for modification are: - Wishes to add/delete/modify/vary functionality - Increase quality attribute - Increase capacity
Artifact	Artifact that needs to be changed: - System user interface - Platform - Environment - System that interoperates with target system
Environment	Modification is done at: - Runtime; by parameter settings - Compile time; using compile time switches - Build time; choice of libraries - Design time
Response	How to make the change: - Locates places in architecture to be modified - Makes modification without affecting other functionality - Tests modification - Deploys modification
Response Measure	How to measure: - Cost in term of number of elements affected, effort or money. - Extent to which this affects other functions or quality attributes

Modifiability encompasses a lot of quality attributes, among which scalability, interoperability, flexibility, maintainability and portability.

Performance
Performance is about how long it takes the system to respond when an event occurs.

Table 3: Possible input for each component in a performance general scenario [BCK03]	
Component	Possible values
Source	Events arrive from: - A number of independent external sources - From within the system
Stimulus	Event arrivals occur by: - Periodic events arrive; for example every 10 milliseconds. - Stochastic events arrive; events arrive according to some probabilistic distribution. - Sporadic events arrive; events arrive at a pattern not captured by either periodic or stochastic.
Artifact	Artifact that is requested for by the event: - System's services
Environment	The operational mode the system is in: - Normal mode - Overload mode; a system state that is able to handle more requests then in normal mode.
Response	The system can respond by: - Processes stimuli - Changes level of service; perhaps the system changes switches from normal to overload mode.
Response Measure	Measuring of how the system responds to arriving events: - Latency; the time between the arrival of the stimulus and the system's response to it. - Deadline; at a particular time a request should be made. - Throughput; for example the number of transactions the system can process in a second.

	- Jitter; the variation in latency. - Miss rate; the number of events that could not be processed because the system was too busy. - Data loss; the number of events that could not be processed because the system was too busy.

Testability
Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.

Table 4: Possible input for each component in a testability general scenario [BCK03]	
Component	Possible values
Source	Testing is done by: - Unit tester - Increment integrator - System verifier - Client acceptance tester - System user
Stimulus	One of the following milestones in the development process that is met: - Analysis - Architecture - Design - Class - Subsystem integration completed - System delivered
Artifact	One of the following artifacts under test: - Piece of design, - Piece of code - Complete application
Environment	The test can happen at: - Design time - Development time - Compile time - Deployment time
Response	The result should be controlled and observed in the following ways: - Provides access to state values - Provides computed values - Prepares test environment
Response Measure	Measuring the result by: - Percent executable statements executed - Probability of failure if fault exists - Time to perform tests - Length of longest dependency chain in a test - Length of time to prepare test environment.

Usability
Usability refers to the ease with which developers can develop when using the framework. This encompasses al kinds of issues during development of an application. Considering that there are no scenario generation tables for, these issues are extracted during the interviews and document with the use of scenarios, just like the other quality attributes. However, the source will be the developer that wants to use an additional service of an IDE, documentation or the framework community. In addition, framework participants can also simplify using other participants thereby also addressing the usability.

Third Addendum

Spring's basic building blocks

The next three sub-paragraphs will elaborate on the role of the following participants:

- Dependency injection
- Spring's Aspect Oriented Programming (AOP)
- Service abstraction layers

11.1.1.1 Dependency injection

A common issue in developing application frameworks, not just in Spring, is how to wire the different elements together. How can, for example, the web tier communicate with objects in the data tier? Spring addresses this problem with a pattern called dependency injection [Fow04]. The basic idea of the dependency injection pattern is to have a separate object, an assembler that populates a field in the listening class with an appropriate implementation. The listening class is the class that is dependent on another class for its correct execution. *Figure 10* shows an example.

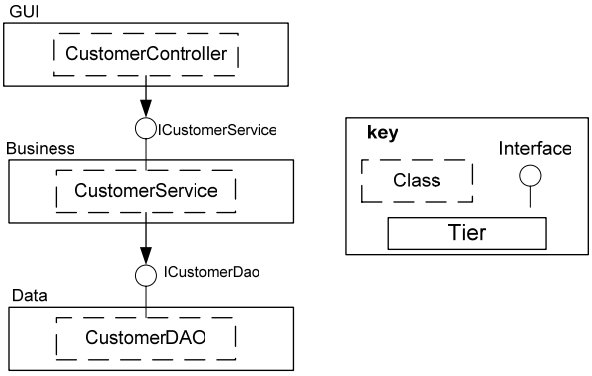

Figure 10: Three tier example with dependency injection

In a class that does not apply dependency injection, the constructor for "*CustomerController*" would look like:

```
private ICustomerService customerService;

public CustomerController() {
        this.customerService = new CustomerService();
}
```

Code fragment 12: Code example without dependency injection

When dependency injection is applied, the code would look like:

```
private ICustomerService customerService;

public CustomerController(ICustomerService customerService){
        this.customerService = customerService;
}
```

Code fragment 13: Code example using dependency injection

As we can see, the constructor now has a parameter of type "*ICustomerService*" that initializes the corresponding field. Subsequently, when a "*CustomerController*" is created, the Spring container will call this constructor with a concrete instance that implements the interface ICustomerService. Spring thereby also promotes programming to an interface instead of an implementation. The Spring framework needs to know

what the dependencies are for each class. This can be configured in code, but configuring in XML files is the expected way. The XML to wire the "*customerService*" into the "*customerController*" for the above code example looks as follows:

```
<bean id="customerDAO" class="com.infosupport.code.example.CustomerDAO"/>

<bean id="customerService" class="com.infosupport.code.example.CustomerService">
        <property name="customerDAO" ref="customerDAO"/>
</bean>

<bean id="customerController" class="com.infosupport.code.example.CustomerController">
        <constructor-arg ref="customerService"/>
</bean>
```

Code fragment 14: Configuring beans by constructor injection

The above example is called constructor injection. Spring also supports setter injection. Setter injection is based on setting the configured dependencies via setter methods in the listener class. The "*customerService*" bean defines a property named "*customerDao*". This means that the "*CustomerService*" class must have a setter method named "*setCustomerDAO*". The container will at start up create the "*customerDAO*" bean and inject the created instance into the created instance of the "*customerService*" bean via the setter method. A discussion on which type of injection to use is outside the scope of this research work and can be read in [Fow04]. In the remainder of the examples, we will use setter injection.

The above example showed the use of dependency injection for classes that are dependent on each other. Dependency injection, as we will soon see, can however also be used for other kinds of dependencies that need to be configured and injected, such as data sources, web services etc.

Dependency injection is also often named inversion of control, see §2.2. However, Fowler [Fow04] argues that each framework uses inversion of control and if these frameworks say they are special because they use it, it is like saying a car is special because it has wheels. Dependency injection is about how classes lookup a plug-in implementation, which can be another class, resource etc. Inversion of control means that the main control of a program is inverted, that is the framework calls application code.

11.1.1.2 Spring AOP

Dependency injection is a first good step in providing a POJO programming model that is unaware of its framework. However, J2EE services as transaction management should also be implemented in the POJOs without them needing to implement certain APIs. The solution is Aspect Oriented Programming (AOP).

AOP provides a different way of thinking about code structure compared to OO or procedural programming. AOP enables us to think about concerns or aspects in the system. Concerns as logging, transaction management and failure monitoring are concerns that are often in numerous classes throughout the architecture. These concerns can be hard to capture. The left diagram in Figure 11 shows the modularization for XML parsing in org.apache.tomcat, which nicely fits in one box, indicated by the red bar. The right diagram shows the modularization for logging in org.apache.tomcat, which cannot fit in one box and can be seen in many places. The problem is that traditional techniques are not able to separate these concerns. AOP addresses this by providing an insertion mechanism to apply aspects, such as logging, to add aspect code to designated locations in the base code[16].

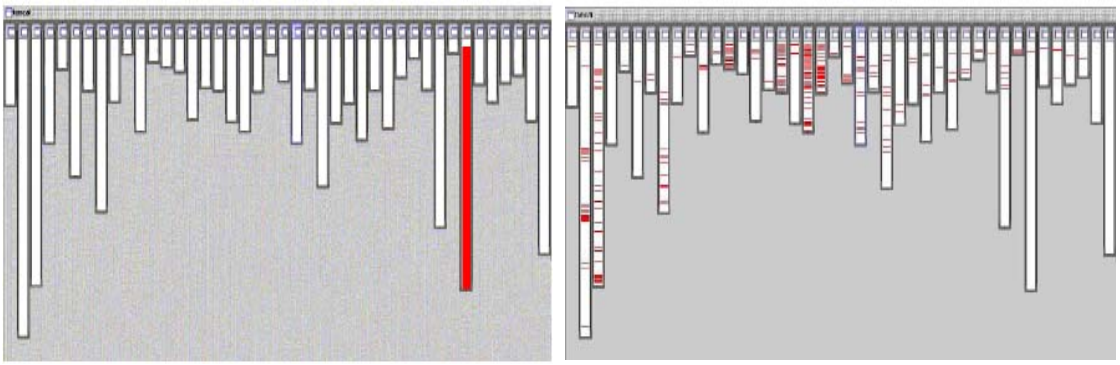

Figure 11: Modularization issue aspects[16]

AOP is a means with which crosscutting concerns, also named aspects, can be injected at places that need them.

Several popular AOP technologies exist. Spring includes an AOP framework that does not require any special classloader that enhances that target class's byte code before the class is introduced in the application. It executes on J2SE 1.3 and above and is using J2SE dynamic proxies or CGLIB byte code generation. The proxies are generated at runtime and will delegate to the target class while applying the aspects. This approach ensures that the POJOs with services applied are portable between environments, including any application server.

Spring also provides support for integration with AspectJ[17], by which requirements as the following can be addressed:

- Negligible performance overhead: AspectJ code is often compiled with a special compiler. It should offer the same performance as writing the aspects manually in the code.
- Non-public methods can be advised: whether this is a good or bad thing, is outside the scope of this paper.
- Fields can be advised: again whether this is a good or bad thing, is outside the scope of this paper. However, in the current author's opinion, fields should only be accessible through setters and getters.

The first goal of Spring's AOP support is to provide J2EE services to POJOs with the dynamic proxies approach. When mapping the scenarios to framework participants a couple of uses of Spring AOP will be discussed.

11.1.1.3 Abstractions from complex J2EE APIs.

Providing a POJO component model by applying dependency injection and Spring's declarative services is not enough. In case of infrastructural issues as connecting to a persistence store, there's no way to avoid working with an API. The J2EE APIs are mostly initially written for usage behind the scenes. JTA, for example, was intended for use in EJB declarative transaction management. Using JTA directly becomes cumbersome because of having multiple exceptions that each need their own catch block. This means a lot of code throughout your application logic, which is abstracted away by Spring APIs.

[16] An introduction to AOP by Professor Paul Klint at CWI and UvA
[17] http://www.eclipse.org/aspectj/

Fourth Addendum

Prioritized scenarios

This Addendum provides the collected scenarios during the interviewing phase. The scenarios are categorized according to the quality attributes they primarily address.

Availability concrete scenarios

1. External component failure		
Nr.	Spring specific scenario	Priority
AV-1.1	The MySQL database server fails to respond. The system identifies the failure, logs it, notifies the administrator by mail and switches to the redundant database. The system will continue to provide its services with a maximum delay of five seconds.	(M,L)
AV-1.2	The external payment web service fails to respond. The system identifies the failure, logs it and notifies the administrator by mail. The system retries until the service is available again.	(C,VL)
AV-1.3	The external payment web service within a distributed transaction, during placing an order, encounters a failure. The system identifies the failure, logs it, notifies the user and rolls back the system to its state before the request. The order is not added to both databases in the former transaction.	(S, VL)

2. Internal component failure		
Nr.	Spring specific scenarios	Priority
AV-2.1	A particular stateless server instance fails within a deployment clustered cluster. The system identifies the failure, notifies the administrator by mail, logs it and directs the request to another server. The system will continue to provide its services with a maximum delay of five seconds.	(C, L)
AV-2.2	Updating an order in a transaction resulted in a database exception. The system identifies the failure, logs it and notifies the user. The order is not updated in the persistence store.	(S, L)

Modifiability concrete scenarios

1. Functionality change		
Nr.	Spring specific scenarios	Priority
MO-1.1	The developer wishes to add functionality to the order service bean. The bean is changed, tested and deployed without the need for recompilation of the system.	(M, VS)
MO-1.2	The developer wishes to change from Hibernate to IBatis. The ORM mapper is replaced, tested and functional within three days at compile time.	(M, L)
MO-1.3	The system administrator needs to change the IP address of a server. The system continues to provide its services with a maximum delay of five seconds.	(S, S)
MO-1.4	The developer wishes to add logging to all existing methods in “CustomerServiceImpl” and “OrderServiceImpl”. The functionality is added at design time.	(M, L)

2. Volume of users change		
Nr.	Spring specific scenarios	Priority
MO-2.1	The system administrator wishes that additional online users are able to connect to the Tomcat web server. The system is adapted at design time. It should be able to handle the larger volume of requests.	(M, VS)

3. Integration with different systems		
Nr.	Spring specific scenarios	Priority
MO-3.1	The developer wishes that an external .NET marketing system can place customers through the customer service by use of web services. The change should be made at design time.	(M, L)
MO-3.2	The developer wishes to set up communication with an external payment system so that customers can pay their orders. This system provides its services through exposing a web service. The functionality is added and tested at design time.	(M, L)
MO-3.3	The developer wishes that an external .NET marketing system can place orders via a *messaging queue*. The change should be made at design time.	(C, VL)
MO-3.4	The developer wishes that when adding a customer to the database it is also added to an external .NET based marketing system. This system provides its services through the use of a *messaging queue*. The system is changed and tested at design time.	(C, VL)
MO-3.5	The developer wishes that an external J2EE marketing system can request orders through the use of *RMI*. The system is changed and tested at design time.	(S, L)
MO-3.6	The developer wishes that when adding a customer to the database it is also added to an external J2EE marketing system. This system provides its services through the use of *RMI*. The system is changed and tested at design time.	(S, L)
MO-3.7	The developer wishes that an external J2EE marketing system's client side, which is build with the Eclipse Rich Client framework, can communicate with Spring bean services via *HTTPInvoker*. The system is changed and tested at design time.	(S, L)
MO-3.8	The developer wishes that when adding a customer to the database, it is also added to an external J2EE marketing system. This system provides its services through the use of *HTTPInvoker*. The system is changed and tested at design time.	(S,L)
MO-3.9	The developer wishes that the order service is exposed as a stateless session bean and deployed to IBM Websphere application server 5.1. The system is changed and tested at design time.	(S, H)
MO-3.10	The developer wishes that when adding a customer to the database, it is also added to an external J2EE marketing system. The system is changed and tested at design time. This system provides its services through the use of a *stateless session bean* that runs in an IBM Websphere application server 5.1. The system is changed and tested at design time.	(M, L)

4. Environment change		
Nr.	Spring specific scenarios	Priority
MO-4.1	The system administrator wishes that the business services and DAOs are deployed to an IBM Websphere application server 5.1. The system is changed and tested at design time.	(C, L)
MO-4.2	The system administrator wishes to change from	(C, L)

	MySQL database to SQL server 7.0. The system is changed and tested at design time.	
MO-4.3	The system administrator wishes to change from Windows to Mac OS X. The system is changed and tested at design time.	(C, L)

Performance concrete scenarios

1. Stochastic events arriving		
Nr.	Spring specific scenarios	Priority
PE-1.1	Events arrive stochastically at the Spring stateless services. For a set of *N* clients the responses need to be handled in maximally *N* seconds.	(C, L)
PE-4.2	Events arrive stochastically at the Spring state full services. For a set of *N* clients the responses need to be handled in maximally *N* seconds.	(C, VL)
PE-4.3	Users read-only requests arrive at the DAOs through the services. The DAOs are able to process the request in *N* percent of the cases in at most *N* seconds.	(C, L)

2. Measuring insight into performance keys		
Nr.	Spring specific scenarios	Priority
PE-2.1	The number of events arriving at the system and the system's response times should be measured. The system administrator has insight into these key performance indicators.	(S, L)

Testability concrete scenarios

1. Class dependencies substituting		
Nr.	Spring specific scenarios	Priority
TB-1.1	A unit tester wishes to be able to unit test the "*CustomerServiceImpl*" POJO without being dependent on the "*OrderServiceImpl*" POJO. The latter POJO is easily replaceable by a mock object during unit testing.	(M, S)

2. Database access methods testing		
Nr.	Spring specific scenarios	Priority
TB-2.1	The developer wishes to be able to test the DAO services without having to communicate with a real database. The DAOs are tested without the need for deployment on a server.	(M, S)
TB-2.2	The developer wishes to be able to test the DAOs without actually modifying the data in the data store. The DAOs are tested with the use of the database, but the data in the database is unchanged.	(M, L)

3. Web service testing		
Nr.	Spring specific scenarios	Priority
TB-3.1	The developer wishes to be able to test the "*OrderServiceImpl*" POJO at deployment time that uses a web service that is not yet available. The application is tested with a temporary replacement of the web service by a mock object without deployment to a container.	(M, L)

4. Rapidly loading unit tests		
Nr.	Spring specific scenarios	Priority
TB-4.1	The developer wishes to be able to run the unit tests of a project that require 10-20 seconds load and start up time without deployment to a container. After the first execution the unit tests should be executed with a start up time within 5 seconds.	(S, L)

Usability concrete scenarios

1. IDE capabilities		
Nr.	Spring specific scenarios	Priority
US-1.1	The developer wishes to rename a class name. The class name is correctly renamed through the use of an automatic refactoring provided by the Eclipse IDE. The system can successfully be deployed to the tomcat web server.	(M, S)
US-1.2	The developer wishes to step through his application code and receive full debug information of the Eclipse IDE.	(M, S)
US-1.3	The developer can use code completion during development and configuration of the beans.	(M, S)

3. Community		
Nr.	Spring specific scenarios	Priority
US-3.1	The developer encounters a problem during development with the framework. The developer has the opportunity to report his problem to a forum. An answer is received within a day.	(M, S)