HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Software Technology

Tuukka Haapasalo

Using Open-Source Solutions in Agile Software Development

Master's Thesis
Espoo, January 17, 2007

Supervisor Professor Eljas Soisalon-Soininen
Instructor Jukka-Petri Sahlberg, M.Sc.(Tech.)

Author: Tuukka Haapasalo

Title of thesis: Using Open-Source Solutions in Agile Software Development

Date: January 17, 2007 **Pages:** xii + 78

Department: Department of Computer Science and Engineering

Professorship: T-106

Supervisor: Professor Eljas Soisalon-Soininen

Instructor: Jukka-Petri Sahlberg, M.Sc.(Tech.)

Software requirements are becoming more and more complicated. Software systems can no longer be developed feasibly without the aid of supporting libraries and tools. For this purpose, commercial software libraries have been created to be reused across multiple projects. Presently, open-source solutions are beginning to emerge alongside their commercial counterparts to support virtually all areas of software development.

Additionally, customers expect custom-made systems to meet their needs in a constantly changing business world. Because requirement changes are on the increase, it is no longer feasible to plan everything up front. Change must be embraced in software development. The relatively new agile methodologies adopt change as an integral part of the software development process.

This thesis studies how well the software libraries, frameworks and tools available from the open source community perform in agile software development projects. A commercial project is initiated to serve as a case study. A set of mature, mainstream open-source solutions is selected for the project, and agile methodologies are used in the project management. The project and the resulting software are measured and evaluated using a set of metrics designed for evaluating both software quality and agile methodology adoption.

On the basis of the results from the metrics and subjective evaluation of the project, it can be concluded that mature open-source solutions and agile development methods can work well together, with features of the frameworks and tools supporting the agile practices.

Keywords: agile, open source, software development, software library, framework, tool, practice

Tekijä: Tuukka Haapasalo

Työn nimi: Avoimen lähdekoodin ratkaisut ketterässä ohjelmistokehityksessä

Päiväys: 17. tammikuuta, 2007 **Sivumäärä:** xii + 78

Osasto: Tietotekniikan osasto

Professuuri: T-106

Työn valvoja: Professori Eljas Soisalon-Soininen

Työn ohjaaja: DI Jukka-Petri Sahlberg

Nykyisille ohjelmistojärjestelmille asetetaan yhä korkeampia odotuksia. Uusien järjestelmien kehittäminen ei enää ole käytännössä mahdollista ilman ulkoisia ohjelmistokirjastoja ja työkaluja. Kehitystyön tueksi on kehitetty monia kaupallisia uudelleenkäytettäviä kirjastoja. Nykyään myös avoimen lähdekoodin ratkaisuja alkaa olla saatavilla käytännössä kaikkien ohjelmistokehityksen osa-alueiden tueksi.

Lisähaasteena ohjelmistokehityksessä on nykyisen liike-elämän tarpeiden vaihtelevuus. Asiakkaat olettavat ohjelmistoprojektien mukautuvan muuttuviin vaatimuksiin myös myöhäisessä vaiheessa kehitysprojektia. Perinteisissä kehitysprosesseissa vaatimukset yritetään määrittää mahdollisimman tarkasti projektin alussa, ja niiden muutoksiin on hankala reagoida. Uudehkot ketterät ohjelmistokehitysmenetelmät asettavat muutoksen kehitysprosessin keskeiseksi tekijäksi.

Tässä diplomityössä tutkitaan, kuinka hyvin avoimen lähdekoodin kirjastot, sovelluskehykset ja työkalut toimivat ketterässä ohjelmistokehitysprojektissa. Työssä käytetään kaupallista projektia tutkimuskohteena. Projektiin on valittu joukko yleisesti käytettyjä avoimen lähdekoodin ratkaisuja ohjelmistokehityksen tueksi. Projektia hallinnoidaan käyttäen ketteriä menetelmiä. Projekti tuloksineen mitataan ja arvioidaan käyttäen hyväksi ohjelmiston laadullisten ominaisuuksien arviointiin tarkoitettuja sekä ketterien menetelmien käyttöönottoa arvioivia mittareita.

Saatuihin tuloksiin ja subjektiiviseen arviointiin perustuen johtopäätöksenä voidaan sanoa, että avoimen lähdekoodin ratkaisut voidaan onnistuneesti yhdistää ketterään ohjelmistokehitysprojektiin. Useat tutkittujen sovelluskehysten ja työkalujen ominaisuudet tukivat hyvin ketteriä menetelmiä.

Avainsanat: ketterät kehitysmenetelmät, avoin lähdekoodi, open source, ohjelmistokehitys, ohjelmistokirjasto, sovelluskehys, käytäntö

Acknowledgements

First of all, I would like to thank my supervisor, professor Eljas Soisalon-Soininen, for his proactive approach and support in getting this thesis done as soon as possible. Similarly, I want to thank my instructor, Jukka-Petri Sahlberg, for his quick responses to my endless questions and for his appropriate, succinct guidance. Thanks go also to my company, HiQ Softplan Oy, for providing me with the setting for this thesis. I wish to especially thank Jukka Paananen for maintaining the good relationship between our company and the client. In addition, I would like to express my gratitude to the staff at GVA Finland Oy for their welcoming attitude concerning this work.

I very much want to thank my beloved Anu for her love and support and also for her sharp eye and detailed comments. I also want to thank my co-worker Jussi Vesala for helping me getting started with thesis writing, and my colleague Jouni Hartikainen for providing me with excellent feedback on my initial draft.

Without the support from all of you, this thesis would not be the same. You have my sincere gratitude.

Espoo, January 17, 2007

Tuukka Haapasalo

Contents

Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
CBO	Coupling Between Object Classes
CK	Chidamber-Kemerer
CMP	Container-Managed Persistence
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
CVS	Concurrent Versions System
DBMS	Database Management System
DIT	Depth of Inheritance Tree
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
EL	Expression Language
FUD	Fear, Uncertainty and Doubt
GoF	Gang-of-Four
GNU	GNU's Not UNIX
GPL	GNU General Public License
GRASP	General Responsibility Assignment Software Patterns
GUI	Graphical User Interface

HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HQL	Hibernate Query Language
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
IoC	Inversion of Control
JAR	Java Archive
Java EE	Java Platform, Enterprise Edition
Java SE	Java Platform, Standard Edition
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JDO	Java Data Object
JDT	Java Development Toolkit
JSP	JavaServer Pages
JSTL	JavaServer Pages Standard Tag Library
LCOM	Lack of Cohesion in Methods
LOC	Lines of Code
LOCC	Lines of Class Code
MVC	Model-View-Controller
NOC	Number of Children
ODBMS,	
OODBMS	Object-Oriented Database Management System
ORB	Object Request Broker
ORM	Object/Relational Mapping
OS	Open Source

OSD	Open Source Definition
OSI	Open Source Initiative
OSS	Open-Source Software
POJO	Plain Old Java Object
POM	Project Object Model
RCS	Revision Control System
RDBMS	Relational Database Management System
RFC	Response for a Class
RMI	Remote Method Invocation
RMI-IIOP	RMI over IIOP
SQL	Structured Query Language
SVN	Subversion
TDD	Test Driven Development
UI	User Interface
WAR	Web Archive
WMC	Weighted Methods per Class
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
XP	Extreme Programming
XP-am	XP adherence metrics
XP-cf	XP context factors
XP-EF	XP Evaluation Framework
XP-om	XP outcome measures

List of Figures

List of Tables

Chapter 1

Introduction

As the software engineering field develops, traditional approaches to software development start to be inadequate. Customers of software development companies have more and more complicated requirements for their software. It is no longer feasible to develop all the components of a system in-house [Altendorf et al., 2002]. Customers expect the software to conform to their needs, even though they are not always able to define them exactly beforehand. Traditional software development processes, in which the customer's first touch to the developed software is in the end of the project, start falling apart. To remedy these problems, new approaches are required [Highsmith and Cockburn, 2001].

Fortunately many software libraries, frameworks and tools exist on the market to aid the software development process. Traditionally these tools and libraries have been proprietary, but more and more extensive and robust solutions are starting to appear on the market from the open source community. Open-source software is software that can be freely used, modified and distributed provided that certain restrictions are observed [Ruffin and Ebert, 2004]. However, for most purposes, the majority of open-source solutions can be used freely, even in a commercial project. Locke [2004] provides numerous proficient arguments for using open-source software in the commercial world, such as low costs, no vendor lock-in, and reliability. Open source projects are guided by technology instead of business needs, with technical excellence often as the primary goal.

A traditional software development process identifies change as the enemy, and sets out to minimise the amount of change required throughout the project. This has, however, proven to be a badly working approach [Highsmith and Cockburn, 2001]. The requirements set out at the beginning of the project are often not final, but change as the project progresses. The initial requirements may be incorrect, even if no apparent change has happened during the project. Presently, eliminating change early means being unable to meet changing business conditions, which leads to business failure [Highsmith

and Cockburn, 2001]. Agile software development methods take an alternative viewpoint to this problem: they embrace change. Instead of trying to plan everything at the beginning of a project, the agile approach is to support change. The agile practices and methods are designed to diminish the cost of change, and to provide early feedback to the customer. Abrahamsson et al. [2002] provide an excellent summary of modern agile methodologies.

1.1 Scope

Both open-source software and agile methodologies are starting to gain grounds in the literature. Studies favouring the adoption of agile practices and open-source solutions have begun to emerge [Hodgetts, 2004; Ruffin and Ebert, 2004]. However, studies concerning the usage of open-source solutions in agile development are scarce. This thesis sets out to study and test *how well a set of mature open-source solutions can be used in a commercial project that is managed using agile methodologies*. This is evaluated by initiating and analysing a case study project.

To accomplish this goal, this thesis studies the current state of software development from the viewpoint of modern Java web applications. The Java programming language provides a solid, robust base for enterprise applications, and Java was a requirement of the case study project. Web applications, on the other hand, are an interesting test subject, because they can be developed rapidly, yet larger web applications require a full-blown server architecture that needs to be comprehensive, scalable and distributable. Additionally, web applications are often usable from anywhere in the world, so they potentially have a very large user group. This thesis concentrates on discovering the areas of web development in which libraries, tools and frameworks can be used. Consequently, software solutions from the open source market are presented to match the requirements. The selection criteria for the open-source solutions is that they must be reliable, widely-used mainstream products with a large user base.

Furthermore, some agile development methods are introduced and studied. In this thesis, the practices of two most popular agile methodologies, Extreme Programming [Beck, 2000] and Scrum [Rising and Janoff, 2000] are studied in detail. Extreme Programming describes a software development approach that consists of a number of common-sense development practices that together form an efficient, yet light-weight process. XP relies on simple initial design and supports even large restructurings later on. Scrum, on the other hand, concentrates on agile project management and not on the used development methods. Again, these were selected because they are presently two of the most popular agile methodologies [Maurer and Martel, 2002; Abrahamsson et al., 2002] and

they can be used together because they target different aspects of the software development process. Additionally, the adoption of agile methodologies is discussed. Hodgetts [2004] reports that it is better to select only a few practices when adopting an agile process instead of trying to tackle everything at once.

To test the selected approach, a commercial case study project is carried out using the presented open-source solutions and a subset of the described agile practices. In the project, HiQ Softplan Oy develops a real estate transaction system for GVA Finland Oy. The resulting software is measured and evaluated with the CK metric suite [Chidamber and Kemerer, 1994], and the adoption of XP practices is evaluated by using the XP Evaluation Framework [Williams et al., 2004a]. Finally, the usage of open-source software in agile development projects is analysed based on both the results obtained from these metrics and subjective evaluation of the project.

1.2 Outline of the Work

The structure of this thesis follows the layout described in the previous section closely. First, chapter 2 describes the current state of web software development and presents several areas in which existing, reusable software solutions could be used. Subsequently, in chapter 3, software solutions from the open source market are identified to meet these requirements. After that, chapter 4 introduces the relatively new agile methodologies and describes the practices of XP and Scrum. Chapter 5 presents metrics for measuring software quality and evaluating the adoption of agile practices. Chapters 6 and 7 describe the project requirements and the selected approach for carrying out the project in detail. After that, chapter 8 goes through the results obtained from the selected metrics and the evaluation of the project. Finally, chapter 9 presents the conclusions based on the executed project.

Chapter 2

Web Software Development

As the software industry has evolved, software projects have got larger and more complicated. Currently, developing a new web-based application wholly in-house is no longer feasible [Altendorf et al., 2002]. Fortunately there are dozens of frameworks, libraries, design patterns and artifact generators that ease the development of new software by providing robust, reliable structure and extensive core functionality to the software.

This chapter presents the current practices for developing new web software. The viewpoint is on Java software development, but the general design ideas apply to other languages as well. First, the traditional JSP web software models are introduced. The terms are still occasionally used and define a basis on which web application design can be built. Subsequently, frameworks are defined and explained. Afterwards, a quick introduction to the Java Enterprise Edition is given. The fourth section describes a typical three-tier application model and lists common requirements for each of the three tiers. After that, design patterns are introduced. Finally, a section is dedicated to automatic artifact generation.

2.1 JSP Models

Early JavaServer Pages specifications defined two models for JSP page access – *Model 1* and *Model 2*. The classifications have been removed from more recent versions. However, the terms are still used now and then, and are explained in [Mahmoud, 2003].

The JSP Model 1 (shown in image 2.1) was intended for small applications. It contains a single JSP page that handles both business logic processing and output rendering. All the request processing, input parsing, business logic and view rendering are handled in the JSP file. While this model supports rapid development it also allows and even encourages bad development practices, such as coding direct access to database [Casal, 2005]. It is therefore not recommended for any production applications, but can still be

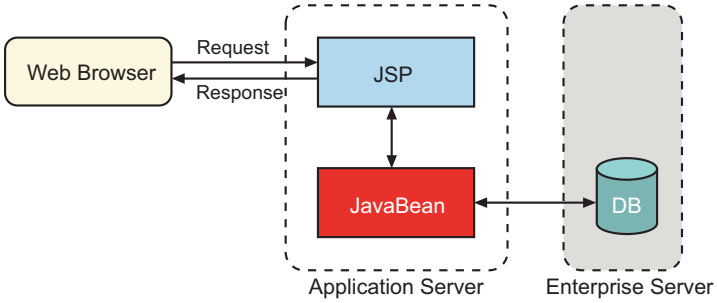

FIGURE 2.1: The JSP Model 1 (adapted from [Mahmoud, 2003])

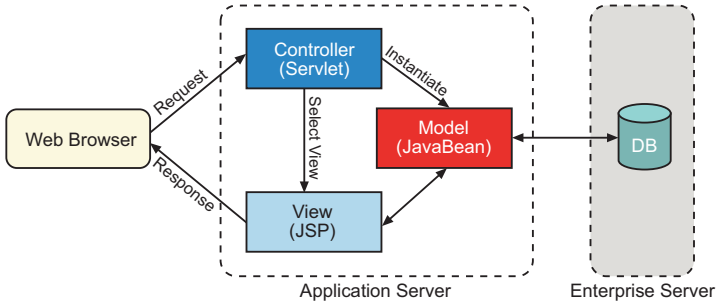

FIGURE 2.2: The JSP Model 2 (adapted from [Mahmoud, 2003])

used in prototyping.

The JSP Model 2 (shown in image 2.1) divides the business logic processing to a controller servlet and the output rendering to JSP pages. The controller reads the input from the user, maintains the state of the system and selects the JSP page that is shown to the user next. This model promotes the use of the MVC pattern (see section 2.4.1) [Mahmoud, 2003], and is the recommended model for modern web applications [Parviainen, 2006].

2.2 Frameworks

When developing new software these days it is very unlikely that the whole software will be written from scratch [Ahamed et al., 2004] – indeed, developing all the components a program requires is feasible only for very small programs. All programs require some functionality that is similar to many other programs; thus, reusable components have been packaged as libraries that can be easily attached to all programs that need them. This is as true for web applications as it is for any other types of applications.

Nowadays the libraries have expanded to full-scale frameworks that support the creation of entire systems. Web-tier frameworks simplify the creation of web-based software

into just modifying the framework configuration files and creating a few implementation classes for processing site-specific logic. Using frameworks and other reusable software solutions may reduce the program development effort to a mere portion of the whole effort – if you are familiar with the aforementioned solutions. It is common that the frameworks involved are hard to learn and have quite a high learning curve, so the increase in productivity might not be apparent at the beginning, maybe not even in the first project. [Johnson, 1997b]

There are several different definitions for a framework; a couple of these say that "*a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact*" and that "*a framework is the skeleton of an application that can be customised by an application developer*" [Johnson, 1997b,a]. The definitions are not mutually exclusive, they just highlight different aspects of frameworks. In contrast to early class libraries, frameworks are not just collections of objects but they define a structure, a design philosophy which the users of the frameworks must follow [Ahamed et al., 2004].

Frameworks can also manage the creation and configuration of all the required objects of the system. This is based on the design pattern called *inversion of control*, which is discussed in more detail in section 2.5.2. The idea behind the IoC adopted by frameworks is that the framework defines the main application structure, and the developers only need to plug their components into it [Johnson, 1997a].

By taking an extensive framework into use, developers do not have to spend their time working on the application infrastructure, but can focus instead on the business logic. In contrast to reusing libraries, frameworks provide also design reuse by supplying a complete, used and tested structure to the application [Parviainen, 2006]. The advantages of using a framework are therefore both code and design reuse, which leads to reduced development times.

However, there are also disadvantages in using frameworks. Because the frameworks define the structure of the application, they restrict the creativity of developers. In addition, since the application structure is tied so deeply into a framework, integrating multiple frameworks can be hard, if not impossible. Another problem with frameworks is that integrating them into an existing application can be tedious. Finally, the performance of an application might not be as good with frameworks as without them. [Parviainen, 2006]

2.3 Java Platform, Enterprise Edition

The Java Platform, Enterprise Edition defines a set of extensions to the standard edition of Java (Java SE) for developing portable server-side Java applications [JavaEE]. Java EE, which was renamed from J2EE with the release of version 5 of the enterprise platform, offers a set of specifications and common interfaces that support building Java enterprise applications. Java EE does not provide concrete implementations for these interfaces but rather lets Java EE vendors provide competing ones [Altendorf et al., 2002]. This thesis adopts the new term Java EE, although all referenced material have used the older term.

Java EE offers a variety of solutions, ranging from an API for XML-based web services through the Java Servlet specification to the transaction and persistence APIs for data storage [JavaEE]. Although there is much hype on the market these days on behalf of Java EE, and the promise is that the productivity of the developers will increase with the adoption of a full-blown Java EE platform, the reality is often harsher. Java EE projects too often deliver systems that are unduly complex and slow. The problem is not in Java EE itself, but rather in how it is used [Johnson, 2003]. There is a current trend of changing the way developers approach Java EE to decrease the application complexity. One of the present ideas is to develop Java EE applications without using Enterprise JavaBeans, which are often seen as the cornerstone of Java EE, but are also the most complex piece of the platform [Arthur and Azadegan, 2005; Johnson, 2003].

2.4 Three-Tier Application Model

Any sufficiently large program is partitioned into separate layers that are in charge of different aspects of the program. These layers are referred to as *tiers* of an *N-tier application model*. The tiers are stacked on top of each other with the lower tier serving the tier above it. A typical adaptation of the N-tier model is the three-tier model, although models with more and fewer tiers are also used [Bertin and Lesieur, 2006]. This section describes the general version of the three-tier application model alongside with common requirements and design propositions for each of the three tiers.

The three-tier application model divides the software into the following tiers [Aarsten et al., 1996]:

Client tier The client tier or the **presentation tier** contains the graphical user interface (GUI) of the program and is responsible for interacting with the user. The client tier does not contain any business logic, but calls the *middle tier* when requested by the user. The middle tier is often separated from the client tier and needs to be reached via a connecting interface (for example, ORB or RMI-IIOP [Aarsten et al.,

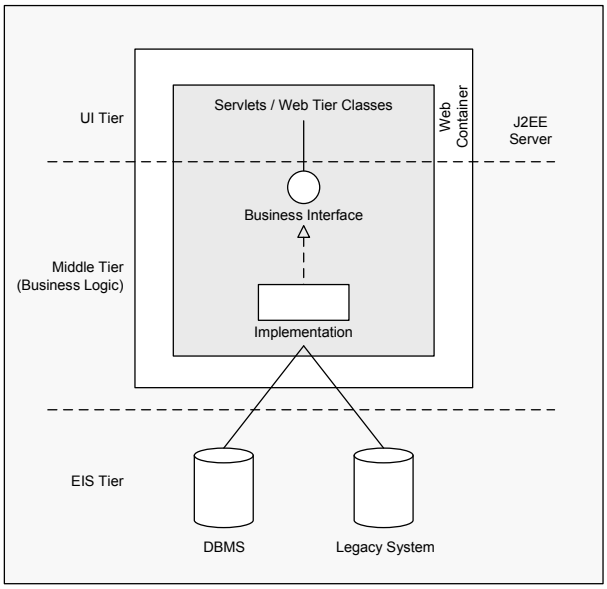

FIGURE 2.3: The Java EE Three-Tier Model (adapted from [Johnson, 2003])

1996; Parviainen, 2006]). However, this is not a requirement, and the middle tier can be reached directly from the client tier.

Middle tier The middle tier or the **business tier** defines the business logic of the system. It contains the business entities that define the domain model of the application, and the business operations that are used to alter the state of the business entities. The middle tier uses the *database tier* to persist the system state.

Database tier Stores and retrieves the business entities used by the middle tier.

In Java EE world, the three-tier application tiers are called the UI *tier*, the *middle tier* and the EIS *tier* (for Enterprise Information System) [Johnson, 2003]. The Java EE three-tier model is shown in image 2.3. The tiers in this model are often distributed to separate machines. Each single tier can in fact be clustered to multiple machines to achieve maximal performance and throughput. Java applications wanting to be Java EE compliant must implement this model as a bare minimum [Arthur and Azadegan, 2005].

2.4.1 Presentation Tier: Model-View-Controller

When designing the presentation tier of an application it is important to keep in mind the principle of *model-view separation*. The model-view separation principle states that the model (the domain representation) of the system should not have direct knowledge of

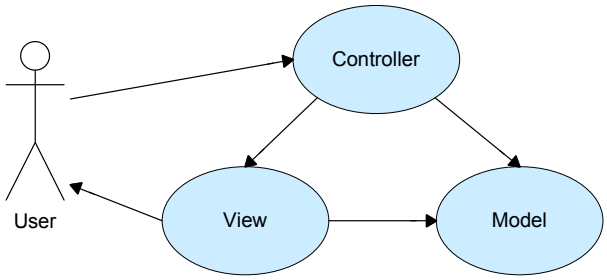

FIGURE 2.4: The MVC Model

the view objects (the presentation tier) [Larman, 2002]. The *model-view-controller* design pattern (MVC) is a model that supports this principle, and is shown in image 2.4.

The model-view-controller pattern divides the program into three pieces: the *model*, which contains the domain model of the system; the *controller*, which coordinates the actions performed on the system; and the *view*, which shows the current state of the system to the user. As shown in image 2.4, the user of the system only gives input to the controller and receives output from the view objects. [Ping et al., 2003; Hussey and Carrington, 1997]

The **model** objects contain both the state of the application and the core functionality of the system. Database access and transactions are also implemented in these components. The model objects offer a stand-alone access to the application data – they usually do not and should not have any direct knowledge of the view or controller components.

The **view** objects provide a visual view of the application state. Everything shown to the user is rendered via the view components. It is important that the view components themselves do not contain any application logic, but only show the current state of the application to the user as represented by the model objects. In the traditional MVC model, the model can inform the view whenever it changes so that the view can be updated. To keep the model-view separation in effect, this is often accomplished with the *Observer* pattern [Larman, 2002], which keeps the model unaware of the actual view objects. In the optimum situation the view object can be easily changed to another implementation without touching the model or even the controller.

The **controller** controls the interactions between the user of the system and the model objects and selects the correct view to show. A standard controller in a request-based system processes the requests, creates the necessary model objects, performs the required actions on the model objects and forwards the user to a view that shows the results of the actions.

MVC is not a new model, and it has demonstrated its advantages by supporting the creation of multiple views to the same data and allowing easy code reuse [Selfa et al.,

2006]. Furthermore the model promotes low coupling and high cohesion (GRASP [Larman, 2002]) by keeping the different aspects of software separate.

Although the MVC model can be used as an architectural model for an entire application, it is often used in the presentation tier of a web application. In a servlet-based Java presentation tier the servlets act as controllers, processing input read from user. The input is then forwarded to the business tier of the application (the model). Finally, the results of the servlet actions are shown to the user via the view components, which are usually JSP pages.

2.4.2 Business Tier: Business Objects and Operations

As discussed before, the business tier of an application contains the business objects and operations that represent and maintain the state of the system. Typically the business objects require a multitude of references to other business objects, helper objects and data sources. A common problem especially in the business tier is the configuration and setup of the required classes. One traditional way of configuring the objects is the usage of the *Singleton* design pattern [Larman, 2002], but the pattern is not without its drawbacks. Using singletons leads to hard-coded dependencies to the singleton objects and each singleton must handle its own configuration independently [Johnson, 2003].

Fortunately application frameworks provide methods for configuring the required objects, usually via XML configuration files. An *application context* object (also known as *registry* or *application toolbox*), creates, configures and manages the required objects. The objects can either be fetched from the application context by specifying their name, or they can be automatically set to all the objects that require them by the application context (see dependency injection in section 2.5.2). This relieves the application developer from the burden of reading configuration files or creating hard-coded singleton objects.

Another commonly heard buzzword in the Java EE world is EJB, the Enterprise JavaBeans, which is also a solution for the business tier. EJB 2.0 defines *entity beans* that represent the business objects, *session beans* that represent the business processes and operations, and *message-driven beans* that coordinate tasks that involve other entity and session beans [Monson-Haefel, 2001]. However, EJBs are a heavyweight technology with significant coding overhead [Altendorf et al., 2002], and their usage is fully justified only when there is a requirement for a distributed environment. In addition, EJB is the most complex technology in Java EE and it is very hard to test because of the distributed environment it requires to run [Johnson, 2003].

EJB 3.0 addresses some of the issues of previous EJB versions but it has just been published and has not yet reached the standard-level. Using it can therefore still be a bit risky [Balogh et al., 2006].

2.4.3 Database Tier: Object/Relational Mapping

Virtually all web software applications operate on some data that needs to be persisted. Most commonly the data is persisted by storing it to a database. Databases come in two distinct varieties: there are relational databases (RDBMS) and object-oriented databases (ODBMS, or OODBMS). The relational databases store textual, numeric and binary data in rows and columns of tables. In contrast, object-oriented databases add data persisting capabilities to the host programming language itself. Even though the object-oriented databases would seem more suitable for object-oriented languages, they have not been widely adopted. For example, all Java EE applications are object-oriented by definition, but still use mostly relational databases. [Johnson, 2003; Bauer and King, 2004]

Using relational databases with object-oriented programming brings out a *paradigm mismatch*: object hierarchies with inheritance and dependencies to other objects need to be stored as row data in database tables. This conversion from an object-oriented world into the world of tables and rows is not a trivial problem to tackle. [Bauer and King, 2004]

A common solution to this paradigm mismatch is *object/relational mapping*. ORM is an attempt to automatically and transparently map the state of objects into the rows of a relational database and back into objects. ORM often requires guidance to govern the transformation. This is provided by supplying and maintaining metadata alongside with the actual objects. Although this implies a certain overhead for the developers, it eliminates the need to write simple create, read, update and delete (CRUD) queries. [Bauer and King, 2004]

In addition to this solution, Enterprise JavaBeans provide their own data persisting capabilities [Monson-Haefel, 2001]. However, as discussed in the previous section, EJBs do have their disadvantages and a more lightweight approach can be more suitable for a non-distributed program [Johnson, 2003].

2.5 Design Patterns

There is no single, clear definition for *design patterns*. Berry et al. [2002] defined design patterns as "*recurring solutions to standard design problems in contexts*." The basic idea behind all the different definitions is that a design pattern describes a common problem and a general solution that can be applied to the same problem in different contexts. A design pattern is therefore a known and widely used solution to a recurring problem. Design patterns evolve from the solutions that are used, they are not invented [Larman, 2002].

In software design, there are patterns in multiple levels. General Responsibility Assignment Software Patterns (GRASP [Larman, 2002]) describe fundamental object design

and responsibility assignment practices. These patterns explain basic ideas such as who should create objects, how they should interact with each other and what kind of references objects should have to each other in object hierarchies. Another group of design patterns is called the Gang-of-Four (GOF) patterns [Larman, 2002]. The GOF patterns apply to the architecture level of a program. The well-known *Observer* pattern, for example, describes how a data item can notify different components (the observers, or listeners) of changes without being tied to their implementations.

This section describes several patterns that apply to web development. One of them, the model-view-controller pattern, was already introduced in section 2.4.1.

2.5.1 Front Controller

The front controller is a design pattern for the presentation tier. It defines a single entry point for requests which enables common control and request handling for the entire application [Berry et al., 2002; Network, 2001-2002]. Centralised request processing ensures that no request processing logic is intermingled with the application views. Sharing the common entry point also enforces a uniform request processing logic for the application.

The front controller pattern contains four different object types:

Controller Provides the entry point for all web application requests. The controller uses *helpers* to process the request and forwards the results to the *dispatcher*. The controller is typically a Java servlet.

Dispatcher Manages the *views* and the navigation between them. The dispatcher selects the proper view and shows it to the user. The dispatcher can be encapsulated in the controller or it can be a separate component.

Helper Responsible both for helping the controller process the different requests and for providing the views with the information they require.

View Used to show the results of the request to the user. The views are often JSP pages, which do not contain any business logic. All information shown on the views is obtained from the helpers.

2.5.2 Inversion of Control

Traditional software library reuse consisted of the developer writing a main program which calls upon the interfaces provided by the reused software components. While this works well with libraries that serve a single purpose and have a clear interface, it also means that all developers must develop and maintain the actual structure of their applications.

The inversion of control (IoC) design pattern provides an alternative approach to program control. With IoC, the reused software library (most often an entire framework) has an application context object that maintains the control of the application and calls on the operations defined by the developer [Johnson, 1997b; Parviainen, 2006]. The IoC pattern embodies the Hollywood principle: "*don't call me, I'll call you*" [Johnson, 2003].

Most current frameworks work according to the principle of inversion of control. By leaving the structure of the application to the framework, the developer can concentrate on implementing the actual business logic.

There are two main types of inversion of control: *dependency lookup* and *dependency injection* [Arthur and Azadegan, 2005]. These are described below.

Dependency Lookup With dependency lookup, the application context provides each managed class with a reference to the context. The objects can then use the API methods of the context to find the resources they need.

Dependency Injection Another approach to managing object configuration is dependency injection. In this approach the context does not supply the managed object with a reference to itself, but instead provides the required references directly. This means that the responsibility for dependency lookup is wholly on the application context, and the objects are freed from dependencies on the application context APIs. [Arthur and Azadegan, 2005]

There are two common ways to accomplish dependency injection. With *constructor injection*, the settings are passed via the constructor of the object. In *setter injection*, the configuration is set via JavaBean properties[1].

2.5.3 AntiPatterns

Although design patterns, when applied correctly, can dramatically enhance the software design quality, they can degenerate to *AntiPatterns* when used in the wrong context or as a solution to the wrong problem. According to Brown et al. [1998], an AntiPattern is "*a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.*" When discussing design patterns, it is important to realise that enforcing the usage of wrong patterns can lead the project to disaster.

As the software engineering industry develops, from time to time certain solutions are presented as the "silver bullet" that will solve all software engineering problems at one go. These software fads include *structured programming*, which was supposed to improve productivity and remove software defects; *networking* technologies, which were

[1] JavaBean properties are class fields that have standard getters and setters. For example, to implement a JavaBean property `colour`, the methods `getColour()` and `setColour()` must be defined.

supposed to make all systems interoperable; and *object orientation*, which was supposed to solve the problems of adaptability and make software highly reusable. While these solutions are not bad, they do not have an answer to all the problems in the field of software engineering. Believing the hype around new software development methods is one of the key factors in turning the methods into AntiPatterns. [Brown et al., 1998]

Several well-known AntiPatterns include **The Golden Hammer**, which happens when a familiar technology or concept is applied to virtually all software problems without regard to its suitability; and **Cut-and-Paste Programming**, which describes the maintenance nightmare situation when code has been reused by copying source code statements. [Brown et al., 1998]

2.6 Artifact Generation

Many programming techniques and tools require information in their own configuration files and in a specific format. Others, like RMI and EJB, demand the creation of interfaces that follow their respective conventions. Creating and maintaining supporting code and configuration files is a tedious and error-prone task. Johnson [2003] reckons that it is impossible to create and maintain EJB 2.0 entity beans by hand when using container-managed persistence (CMP), because they are far too complex.

To tackle the issues with supporting code and configuration files, tools that generate the required artifacts automatically have been created. Code generators can create all the classes required by the EJB architecture from a single file. Configuration file generators can create framework configuration files from source code that is annotated with special annotations. These tools help reduce the amount of code required and remove the separation between the definition of an object and the required supporting configuration.

Code generation in itself can mean at least two things: generating code automatically from special markup; or generating code from formal specifications. While code generation from formal specifications can be seen to provide the most benefits, such as dramatically reduced development effort and provably correct implementation (with respect to specifications) [Whalen and Heimdahl, 1999; Ngolah and Wang, 2004], many of these benefits apply to code generation from markup also. Automatic configuration file generation ensures that there are no syntactical errors in the files. Extracting interfaces automatically eases the creation and maintenance of EJB entity beans [Johnson, 2003].

2.7 Summary

This chapter presented several key concepts in modern web software development. The introduction to web development started with the JSP models, which are now taken out of JSP specifications. However, the JSP model 2 can still be seen as a valid starting point for designing web applications.

Section 2.2 identified the need for using frameworks in web development. Frameworks provide a consistent structure to the application as well as supplying robust and tested functionality.

The Java EE platform was introduced in section 2.3. Java EE provides a set of common interfaces for building enterprise applications. Although it is widely used, many projects have produced unduly complex systems when using Java EE in the wrong way.

Section 2.4 presented the three-tier application model, which consists of the *presentation tier*, the *business tier* and the *database tier*. This model is the most typical enterprise application model and is the smallest model that can be considered Java EE compliant. Software needs were identified for each of the three tiers. For the presentation tier, a model-view-controller framework is recommended. In business logic tier, a business object framework is required. For the database tier, support for the object/relational mapping is needed.

In section 2.5, some important design patterns for web development were introduced. These include the *front controller* pattern, which is used as a single point-of-entry in the presentation tier of a web application; and the *inversion of control* pattern, in which a framework is responsible for program control.

Finally, section 2.6 described the need for automatic generation of various software artifacts. Automatic artifact generation reduces implementation effort and removes syntactical errors from the generated artifacts.

Chapter 3

Open-Source Web Development Solutions

The previous chapter identified several areas of interest in web development. In this chapter open-source solutions for those areas are presented. The chapter begins with an introduction to open-source software. After that, the usage of open-source software is justified. Finally, the selected open-source solutions are presented.

3.1 Introduction to Open Source

Open-source[1] software, in contrast to proprietary software, is software that must have its source code available for developers to see, improve and to create derivative works from. Open source is not shareware, which is mostly inexpensive proprietary software that can be tested freely before buying it. On the other hand, open-source software does not have to be free, in the meaning that users would not have to pay for obtaining it. The defining characteristics of open-source software are that users must be able to freely use, redistribute, modify and distribute derivatives from the work, under the terms of the original license [Locke, 2004]. There are other requirements for open-source software, but those are not covered in this work.

The full definition of OSS is that it must meet the Open Source Definition (OSD), which is maintained by the Open Source Initiative [OSI]. OSI is a non-profit corporation that promotes the usage of open-source software in the business world. In theory, anyone can contribute to the development of an open-source program. This is a radical change from the proprietary, in-house software development style. In the OS development model the software that is being created can be inspected by thousands of devel-

[1] [OSI] suggests that when using the term *open source* as an adjective, it should be hyphenated.

opers from all around the world, instead of the selected few developers of a proprietary software. According to OSI, this reduces the development time of the software and improves the quality. When the software is exposed, the software defects can also be found faster.

For developing software for your own use, such as creating a web application that is installed on one of your own servers, there are usually no restrictions on the use of open-source software. OSS licences most often only apply when you are distributing the software. For example, the common GPL license places requirements for you only if you create a software derived from or added to a GPL software that you *actually distribute to someone* [Locke, 2004]. Nevertheless, the license terms must be examined carefully when taking open-source solutions into use [Ruffin and Ebert, 2004].

3.2 Using Open-Source Software

Open-source software is generally surrounded by myths and beliefs that it cannot really compete with proprietary software. These beliefs are referred to by open source proponents as FUD for the fear, uncertainty and doubt that they generate [Locke, 2004]. It is feared that the quality of the open-source software is not high enough, that it is not documented, not standard and not mature enough. Another fear is that the development and maintenance of a selected open-source software stops, and soon the software cannot be used anymore because of unfixed bugs or new technologies that are not supported by the software.

Unfortunately, new Internet technologies are often buggy and non-standard, and this certainly applies to open-source software as well [Altendorf et al., 2002]. However, there is one point about the quality of open-source software that is not often thought of: the OSS development process is not tied to deadlines and budgets that haunt the proprietary software development process. Instead, OSS development is driven by the technology itself. Often the primary goal of an OSS development project is technical excellence [Locke, 2004]. In comparison, proprietary software always has business interests at the background, and sometimes those business interests drive unfinished software to the market to meet the deadlines.

While it is possible for a very small open source project to die and leave its users stranded with a piece of software they cannot use, a mainstream open-source software that has a large user community is a lasting solution that can be safely taken into use [Ruffin and Ebert, 2004]. If developers leave the OSS development project others will take their place.

Many people fear that taking an open-source software into use restricts the project into

using only open-source software [Locke, 2004]. In truth, almost everyone is already using open-source software, perhaps without even realising it. The best-known open-source software is the Linux operating system, which has surpassed the various proprietary Unix operating systems in the server operating system market [Dedrick, 2004]. Another well-known OS software, the open-source Apache Web server [Apache], is used in over 60% of the current web server installations [Netcraft, 2006].

3.3 Open-Source Solutions

This section presents solutions for the different areas in web development that were identified in chapter 2. In addition, some OS solutions for other development supporting purposes are introduced. The solutions given here represent *de facto* standard, mainstream open-source solutions available on the market currently. The focus is on production-quality, mature software rather than bleeding-edge development versions of OSS.

3.3.1 Presentation Tier: Apache Struts

For the presentation tier the Apache Struts framework is described [Struts]. Struts is a web framework that supports the creation of a web presentation tier. From late 2002, Struts became the natural choice for Java EE web applications [Johnson, 2005a]. Struts is still the standard choice for the presentation tier in web applications, it is widely used and has a large user base [Casal, 2005]. However, it is speculated that this is only due to it being the first proper web framework on the market, and that more recent frameworks will take its place somewhere in the future.

Apache Struts implements the MVC design pattern with a slight modification to support web usage. In contrast to the traditional MVC model, where the model components *push* the information to the view components, a web MVC view component must *pull* the information from the model while being rendered [Johnson, 2005a]. In Struts, the MVC parts are the following [Goodwill, 2002]:

View The JavaServer Pages that render the HTML pages shown to user

Controller The `ActionServlet` class, which forwards the request to `Action` subclasses

Model Struts does not provide any model classes; these must be provided by the application developer

Apache Struts functions around a single front controller, the `ActionServlet`, which delegates all the requests coming to the web application. The controller is configured by a configuration file to forward the requests to specific actions, which process them based

on the input data. The actions can select the resulting page via string-keyed mappings – for example, an action might process the request and return a mapping with the key "failure", which in turn can be configured to point to a specific view page.

In addition, Struts supports concentrated exception handling by creating a subclass of `ExceptionHandler` provided by Struts and attaching that to the desired actions via the configuration file. This allows handling all the exceptions of the entire presentation tier, which in turn makes the code of the actual actions simpler.

In Struts, the HTML input forms that are shown on application views are mapped into `ActionForm` components, which must define the fields of the form as JavaBean properties. Struts automatically loads the values sent by the user to the `ActionForm` class, where they can be easily fetched from. Struts also contains a *Validator* component that can be configured to automatically check the input values both on the client side and on the server side. The Struts Validator has been later on moved to the Jakarta Commons [Commons] project, which is a repository of reusable software libraries.

The actual (X)HTML output can be generated with any suitable techniques. However, Struts provides supportive tag libraries for JSP that extend the standard JSTL libraries, and supports using the JSTL Expression Language with Struts EL. For example, localised error messages can be rendered to output pages with the HTML tag library function `<html:errors />`.

The main advantage of Struts is its structure, which is also its main weakness. Basically, all Struts applications must be coded in a similar fashion. Enforcing a given architecture keeps the code at least above a certain level, but on the other hand, it also restricts the freedom of the developers.

3.3.2 Business Tier: Spring Framework

The Spring Framework [Spring] is a Java EE framework that is based on code published in [Johnson, 2003]. It was first released as an open source project in January 2003, and it has quickly become the dominant Java EE application framework [Johnson, 2005a]. While Spring might provide less services than full-blown frameworks, Arthur and Azadegan [2005] see its lesser complexity as a way to increase productivity. Spring provides a way to create Java EE compliant software without using EJB.

Spring is a layered framework, which means that developers can choose the parts they need in their own application. At the core of Spring is the inversion of control container and bean factories, which can be used to create and configure the objects required by the application. The creation and configuration of these objects, or beans, is based on XML configuration files. All beans under Spring have a logical name that is used to setup the object configuration. The beans can also be fetched in application code from

the Spring container with the logical names. In addition to managing references to other beans, variables of normal primitive types can also be set via the configuration files. This removes the need to have hard-coded singleton configuration objects in the application code. Perhaps the simplest way to integrate Spring to an application is just to move the application object management under Spring configuration. [Johnson, 2005b]

However, Spring also provides many other features. For example, the JDBC data sources of an application can be created, configured and injected to objects that need them by Spring. Spring does not, however, reinvent the wheel. There are no database connection classes in Spring, but the standard components of the developer's choice can be configured via Spring configuration files. Spring supports integration to popular data access technologies, such as Hibernate, JDO, Oracle TopLink and iBATIS.

For basic JDBC access, Spring provides wrappers that change the checked exceptions of JDBC into unchecked ones. The design idea behind this practice is that in most cases an application cannot do anything about a database exception. In this situation it is useless to force application developers to catch all exceptions. JDBC is one of the few data access APIs that still use unchecked exceptions. TopLink, JDO, and the newest version[2] of Hibernate use unchecked exceptions exclusively. In case of a recoverable exception, developers can still catch the unchecked exception, they just are not forced to do so [Johnson, 2005b].

Spring also supports aspect-oriented programming (AOP), has its own web framework (the Spring Web MVC) and is developed to support easy testing. The Spring container can be initialised in a single line of code to be used in the entire application as well as in unit tests.

3.3.3 Database Tier: Hibernate

The database tier requires communication with the actual databases. Traditionally this has been accomplished by writing long queries with the Structured Query Language (SQL). Hibernate [Hibernate] provides an alternative, light-weight approach: mapping the model objects into database tables by simple configuration files or by annotating the code itself with hints on how to store the fields and hierarchies of objects. Hibernate is not the only solution for the ORM issue, but it was the first popular, fully featured and open-source solution [Johnson, 2005a], and it has grown in popularity.

Hibernate allows developers to write the data objects as Plain Old Java Objects, or POJOs, rather than force the usage of EJB with its remote and local interfaces. The POJOs are mapped either by writing configuration files or by annotating the code itself. *Hibernate Annotations*, which was just released, supports using Java 5 annotations

[2] Hibernate switched from checked to unchecked exceptions in version 3.

to specify the object/relational mapping for the persistent objects.

The mapped objects can be stored to and fetched from the database via simple methods such as `session.get(class, id)` and `session.saveOrUpdate(object)`. More complex queries can be written with Hibernate's replacement for SQL, the Hibernate Query Language. HQL is similar in syntax with SQL, but references to objects and fields are written with the names of the actual classes and their properties instead of the database tables and columns. In addition, the HQL parser obtains information about the actual class hierarchy from the mapping configuration, and uses that to fill in missing query information. For example, the following statement is a perfectly valid HQL query statement:

```
from User as user where user.company.name = 'Firm'
```

Object hierarchies are mapped automatically, with no need for developers to query them manually. In a full-blown application that uses Hibernate it is perfectly normal that there are no SQL queries and only a couple of HQL ones (mostly for searches). All the basic work will be done with the mapping specifications. However, Hibernate does allow the usage of standard SQL too, for those situations where HQL is not enough or a non-standard, vendor-specific database extension is required. In fact, developers can override all the queries that Hibernate automatically constructs with custom SQL ones.

3.3.4 Artifact Generation: XDoclet

For generating source code, the XDoclet engine is presented. XDoclet is a widely used, extensible code generation engine [Tilevich et al., 2003]. Currently there are two different versions of XDoclet, XDoclet 1 [XDoclet1] and XDoclet 2 [XDoclet2]. XDoclet 2 is a rewrite of XDoclet 1, and these are developed and maintained separately.

XDoclet is an artifact generation engine that supports generating code and artifacts for many existing technologies. These include EJB, Hibernate, JBoss, Struts and Spring Framework. The artifacts are generated based on XDoclet annotation tags (which resemble JavaDoc tags). Usually XDoclet code generation is tied to the build process by invoking XDoclet via Ant tasks.

Using XDoclet can reduce the amount of code written dramatically. For example, a simple EJB bean can consist of seven different files which need to be maintained. With XDoclet it is possible to maintain a single file, from which the rest are generated. [XDoclet1]

3.3.5 IDE: Eclipse

For the development environment, the Eclipse platform [Eclipse] is introduced. Eclipse has been gaining popularity ever since its creation, has a very large support group and is now the Java IDE of choice for the open source developer [Arthur and Azadegan, 2005; Geer, 2005].

Although Eclipse is actually a rich client platform that supports the creation of any software (for example, a BitTorrent client called Azureus is developed on top of Eclipse), the Java Development Toolkit is the IDE that is most often associated with the word Eclipse.

The current version of Eclipse's Java Development Toolkit provides full support for the Java language, including the Java 5 extensions (generics, annotations, and so on). The JDT contains numerous time-saving features such as code completion, class import management, easy navigation between classes, searching for implementing classes and subclasses, automatic JavaDoc showing and multiple refactoring tools. Refactoring tools include the generation of getter and setter methods for class fields and updating the calling code when renaming or moving classes and methods.

3.3.6 Building: Apache Maven

Any complex project requires a build tool. In traditional C development, the Make tool has been the official tool for decades. There are, however, shortcomings in Make with cross-platform Java development, and so the Ant tool was developed and accepted as a substitute for Make in the Java world [Serrano and Ciordia, 2004]. Even though Ant has many improvements over Make with cross-platform support and custom build tasks, it is still based on the same idea as Make: both are configured with long configuration files that contain instructions on running all of the required build commands. Apache Maven [Maven] is a popular open-source build tool based on a different ideology. Maven favours the idea of *conventions over configuration*, which means that build settings, such as source code directories and build directories, can be shared between projects. In fact, Maven defines a standard directory layout for projects. Those projects that adhere to the standard layout do not need to configure their application paths at all. A new project only needs to initialise the Maven directory structure with a single Maven command, and fill in the automatically created project object model (POM) configuration file. In the simplest case, only the name of the project and the distribution packaging type (for example, JAR or WAR) needs to be filled in. After that, source code that is placed in the correct directory can be automatically built, tested, packaged and distributed. [Maven; Smart, 2005]

Maven also introduces a new way of handling project dependencies. Required JAR packages are often kept under version control only for distributing them to all project developers. There are no real reasons for version controlling the packages themselves. With Maven, the packages required by the project are kept in one or multiple Maven repositories. The requirements, or dependencies, are written to the Maven configuration file. Maven will then automatically download the required artifacts from the repositories. Maven provides a central repository that contains a number of standard Java project requirements, such as logging components and the Apache Commons [Commons] components. In case the required artifacts are not found on the central repository, a company can easily set up its own Maven repository. In addition, when running Maven tasks, the dependencies are automatically appended to classpath, so there is no need for developers to maintain long lists of classpath entries in different configuration files. [Maven]

3.3.7 Version Control: Subversion

Any software project that is going to production requires a version control to store the different versions of the software and to help merge conflicts that happen when two developers modify the same source code file at the same time. The dominant open-source version control system has been the Concurrent Versions System CVS. However, CVS and RCS, which it is based on, are old technologies, and they have not changed to match new software development requirements. [Chu-Carroll et al., 2002]

A new, compelling open source replacement to CVS is Subversion [Subversion]. It has been developed to be a replacement of CVS, so developers working with CVS should have no difficulties in moving to Subversion. Most of the problems with CVS have to do with directories, since RCS does not know anything about them. Subversion has been developed to fully support directories and to maintain version history when moving files around.

Branching in Subversion has also been changed from the traditional style. There are no conventional (CVS-style) branches in Subversion; rather, any directory can be virtually copied under a new name. The new directory is just a link, and will store only the modifications that are made under that directory. The version history before the copying point remains the same for both directories. These virtual directory copies effectively work as branches, but they can also be used for other purposes.

Subversion can be run either as its own server or under the Apache HTTP server. When running under Apache, the Subversion code repositories can also be reached via normal HTML web browsers. This allows for easy code and change inspection from outside the office.

3.4 Summary

This chapter provided an introduction to open-source software and the Open Source Initiative, which maintains the Open Source Definition and promotes the use of open-source software in software business industry.

Section 3.2 supplied the motivation and justification for using open-source software. While some open source projects can be of questionable quality, most of them compete with their proprietary counterparts. For some open source projects, technical excellence is the primary goal. Using mainstream OSS can be seen as a safe choice.

In section 3.3, various OS solutions were presented for the problems and requirements identified in chapter 2. The section listed well-known, mainstream OS solutions that can be taken for production use.

Chapter 4

Agile Software Development Methods

Software developers are beginning to realise that traditional, waterfall-style development methods are starting to fall apart in the rapidly changing business software development field [Spence, 2005]. The waterfall model, which consists of all the standard software development phases lined up one after the other [Schach, 2002], is the oldest software life-cycle model and has been widely used in both large and small projects [Huo et al., 2004] with successes especially in large and complex projects. However, at the very core of the waterfall model is the idea of planning everything at the beginning of the project. Change is not an aspect that is anticipated in waterfall-based development, and change is what the modern business software development is all about. Trying to decide everything as early as possible means being unable to adapt to new business conditions, which leads to business failure [Highsmith and Cockburn, 2001]. In the present situation, the waterfall model just does not work [Spence, 2005]. There is a demand for new models that allow developers to change the plans even later on.

In this chapter, the relatively new agile approach to software development is presented. First, an overview of the agile processes is given. Then, the two major agile processes, Extreme Programming and Scrum are introduced in consecutive sections. Finally, an approach for adopting agile methods is proposed.

4.1 Overview

The need for models that embrace change has been identified. Most of the traditional waterfall-based software development projects, which have been planned as far as possible, still have to change. Highsmith and Cockburn [2001] report that in a study of more than 200 software projects, nearly half of them did not have their original plans in use at

the end of the project. Conforming to the plans had not been the primary goal anymore, and the projects had to respond to requirement changes instead.

Agile development methods promise to bring a solution to this need. They take a different perspective to software development in comparison to the waterfall model. The basic lesson in software engineering has been that the cost of change increases exponentially as the software moves towards production use [Schach, 2002]. The motivation has been to identify and carry out changes as early as possible. However, with a proper combination of technology and programming practices, it is possible to stop the cost of change from increasing [Beck, 2000]. In agile software development, responding to change is emphasised.

Accepting change and allowing large-scale changes later on in a project is not enough, however. If the cost of changes still rises exponentially as the project proceeds towards production, changing requirements at a later phase can still lead the project to a disaster [Beck, 2000]. The possible solutions must supply a way to stop the increase of the cost of change, or at least to slow it down. In the waterfall model, the cost comes from having to redo the entire development cycle. To carry out a requirement change to a software that is in its maintenance phase, a developer must change the requirements, redo the specifications and design, implement the change and retest the entire application to ensure that everything is still working [Schach, 2002]. The cost, therefore, comes from having to maintain a multitude of documents and having to carry out extensive and expensive tests. Implementing the change might not be easy either, if the architecture has not been developed with changeability in mind. Agile methods respond to these requirements by requiring less documentation, making the software easier to change and by having automated unit tests to support software changes.

4.1.1 Documentation

For documentation, the agile viewpoint is that there should only be the bare minimum of it. Cockburn [2002] sees software development as a game that has the production of the actual software as its primary goal. Maintaining extensive documentation is a primary requirement only for setting up the next game, as the updated documentation is only needed to make it possible for new developers to understand the system. In the gamer's perspective, setting up the next game is only a secondary goal. If the primary goal is not reached, the game ends in any case. Therefore, there should only a *sufficient* amount of documentation.

4.1.2 Testing

Testing is another important aspect of agile methods. In contrast to the waterfall model, which places testing at the end of the development cycle, most agile methods advocate test-driven development (TDD), which states that tests must be written before the code itself [Beck, 2000] and that a failing test must be written before a defect is fixed [McBreen, 2003]. In addition, every program feature must be covered with tests. These tests are written as automated unit tests, which should always run at 100% success, except during the time when implementing a new feature or fixing a defect [Beck, 2000]. These tests provide rapid feedback on the effects of a change.

The XP method (see section 4.2) suggests that unit tests and functional acceptance tests are sufficient for a project [Beck, 2000]. While these tests are not necessarily as good as tests created by professional testers, organisations adopting XP have discovered that the quality of the software itself is better [McBreen, 2003]. Test-driven development is one of the essential XP practices, which should always be included even when adopting the practice incrementally [Auer and Miller, 2002]. Hodgetts [2004] reports an immediate decrease of defect counts, from 20–30 to only a few per iteration, with the adoption of test-driven development.

4.1.3 Simple Design

To keep the code easy to change, agile methods favour a simple design. The idea is to develop the simplest solution that could possibly work for the current feature [Beck, 2000]. Since the future is uncertain, it is useless to try to design the system as it is at the end of the project. If and when the design solution is not good enough anymore, it is *refactored*[1] to meet the new requirements. Refactoring must also be used to keep the software structure as simple as it can be and to remove duplicated code. This idea of *designing through refactoring* keeps the code clear and allows developers to design only what is required to implement the next feature. [Beck, 2000]

Without the extensive unit test cases that need to be built, the developers might be unwilling to perform the required refactorings to keep the code simple. However, the unit tests provide the safety nets that are required to assure the developer that everything is still working after the change is done. [McBreen, 2003]

[1] Refactoring means changing the current internal structure of the code without changing its external behaviour [Auer and Miller, 2002].

4.1.4 Software Quality

Because the design is simple, not planned totally up front, and the documentation is light, the quality of the produced code can be seen as questionable by traditional waterfall model proponents [Huo et al., 2004]. However, the agile methods do have quality attributes.

First of all, as noted before, test-driven development has been seen to reduce the amount of simple defects [Hodgetts, 2004] and simple design that is frequently refactored reduces unnecessary code complexity and maintains higher code quality [Spence, 2005]. Agile methods also include *continuous integration* (see section 4.2), which means that code is integrated to the code base as soon as it is ready [Abrahamsson et al., 2002]. This reduces the defects introduced with massive infrequent integrations. In addition, the acceptance tests can also be automated [Auer and Miller, 2002]. In theory, all the tests a software system needs could be run with a single command. In conclusion, with agile methods the quality checking practices happen with a higher frequency than in the waterfall development method, and they are available earlier [Huo et al., 2004].

4.1.5 The Agile Alliance

In early 2001, seventeen advocates of lightweight development processes gathered together in Utah and formed a group called the Agile Alliance [AgileAlliance; Cockburn, 2002]. In their meeting they produced the Agile Manifesto [Beck et al., 2001], which lays down the basics for agile development methods. The manifesto presents the viewpoint that agile development should have. The values of agile manifesto are listed below. The manifesto acknowledges the values of the items on the right side, but holds the items on the left side more important.

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

The values emphasise the human role as opposed to industrial processes. A working team spirit is important in the agile methods. The second point underlines the importance to continuously produce new, working versions of the software, with short release periods. The third point sets great store by creating good relationships with the customer instead of negotiating strict contracts. Finally, the ability to make changes to the software even later on in the project is highlighted. [Abrahamsson et al., 2002]

4.2 Extreme Programming

The development of XP started from the problems noticed with the long development cycles in the traditional software development models. The first motivation was to create a method for simply getting the work done, and the theoretical basis was established after a number of successful trials in practice [Abrahamsson et al., 2002]. XP is a lightweight method that is designed for small-to-medium sized teams developing software with rapidly changing requirements [Beck, 2000]. XP has been the most popular agile methodology [Maurer and Martel, 2002], and has been widely accepted as the most important one [Mahnic and Drnovscek, 2005].

XP has been very successful in its area of applicability [Cockburn, 2002], and mostly successful experiences have been reported when using it [Abrahamsson et al., 2002]. Programmers like to develop with XP, since in XP they get to do more actual programming work [Beck, 2000]. However, McBreen [2003] sees a big problem in this. Since developing with XP is more fun than doing things the traditional way, developers inflected with XP might not want to change back to using the traditional methods when their usage is required, for example with large and complicated projects that have well-known requirements.

4.2.1 Practices

At the core of XP are twelve simple practices, which are listed in this chapter. None of these practices are new in software engineering [Beck, 2000]. The promise of XP is to bring them together in a way that produces the most benefits. The *extreme* in Extreme Programming comes from taking these standard practices to extreme levels [Abrahamsson et al., 2002]. The practices listed here have been taken from [Auer and Miller, 2002; Beck, 2000; Cockburn, 2002; Maurer and Martel, 2002].

Planning Game The planning game in XP is XP's response to the problem that you cannot know everything beforehand, especially in software engineering. The main idea in planning game is to form a rough plan of the next iteration quickly, and then refine it as the iteration moves on. The planning game is a collaborative game with the customer and the developers working closely together. The developers participate by estimating how much implementing a software feature will cost, and the business people must decide which features will be included to the iteration and what are their priorities.

Short Releases XP emphasises that new versions of the software should be released often, in iterations of one or two months, rather than once or twice a year. Frequent

iterations provide early value to the customer. In addition, the customer is quickly able to verify that the developed software is working as required, thus reducing customer risk and providing valuable feedback to the development team. The iterations must still produce complete features.

Metaphor A metaphor in XP is a shared story between the customer and the developers. It creates a common vocabulary and guides the developers when designing the software. The XP metaphor replaces the architecture in traditional software. An example of a metaphor is, for example, an assembly line, which could be used to describe customer help desk. Phone calls could be seen as items that are passed through different persons (working at the assembly line). This provides a way to look at the software at a familiar angle, and helps raise questions such as "what happens when the item (phone call) reaches the end of the line?"

Hodgetts [2004] reports of a project that was having problems with certain interface code that did not have any consistent structure. They identified the lack of any guiding metaphor as the problem. As a solution, a metaphor was sought and found, and the existing messed-up interface code was refactored a bit at a time to match the new metaphor. After a few iterations the interface code had been cleaned and implementing new features was easier.

Simple Design Generally software design is "implemented for today and planned for tomorrow" [Beck, 2000]. XP states that since you cannot know the future, you should not try to make extensive plans for it. The design of a software must always be as simple as possible. This means that the software must run all tests, have no duplicate logic, state every intention that is important and have the fewest possible classes and methods. The software must not contain any functionality that is not currently needed, because the need for that functionality might never arise.

Testing XP contains two types of testing, namely, unit testing and functional acceptance testing. XP applies test-driven development with the developers writing the unit tests, and the customer provides the functional tests. All unit tests must pass at all times (except during the actual development of a new feature or when fixing a defect). Functional tests, on the other hand, can fail, as they are used to measure the development progress. When all functional tests complete successfully, the software is ready. Testing is one of the few necessary aspects of XP, even when adopting the method incrementally. If you do not write tests, you are not being extreme [Beck, 2000].

Zhang et al. [2006] report that projects using TDD have produced software that is more reliable, easier to maintain and more efficient, and have produced it up to 10%

faster than with traditional methods.

Refactoring Refactoring is the process of changing the internal structure of the code without changing its functionality. Refactoring is applied to remove duplication, to improve communication, to simplify the design and to add flexibility. When adding a new feature, developers look at the code and think of ways of changing it so that adding the feature is easier. After adding the feature, developers can look at the code and think of ways to simplify the resulting design. These are both forms of refactoring. Because refactoring often requires changes to many parts of the system, extensive unit tests are needed to assure the developer that everything is still working afterwards. While a single structural change is usually quite small, multiple consecutive refactoring changes can be used to achieve large restructurings with less chances of breaking the application code [Huo et al., 2004].

Pair Programming In pure XP, every line of code is written in pairs, with two developers working on one computer. The one holding the keyboard and mouse is reflecting on the best way to implement the actual method under work, and the other one is looking at the general view, thinking of the consequences and possible simplifications. The pairs switch their roles every now and then. Although the method may seem inefficient, it provides several important benefits. Every part of the system is familiar to at least two developers, and when working as a pair, the developers motivate each other to better development practices. The developers pair with different people frequently, thus spreading the knowledge of the system inside the team.

Most experiences with pair programming have been positive. Williams et al. [2000] report that developers who have used pair programming have worked harder and smarter, and have produced results early. Hodgetts [2004] describes that a team considered pair programming too wasteful of resources at first, but later on developed the motivation for using it. In that team pair programming had helped spread out expert knowledge from a single developer and had thus removed a bottleneck.

Collective Ownership Collective ownership means that each developer is allowed to change any part of the code at any time. Having the right to change code is a basic requirement to be able to apply refactoring. Without control, this could lead to chaos. This is mitigated in XP by making everyone responsible for the entire code, and by having the unit tests that should run at all times. If developers break something, they must fix it so that the unit tests run again.

Continuous Integration Code is integrated to the version control system frequently, at

least once every day. Each pair integrates their modifications, and after the integration, all unit tests should run. This removes the integration nightmare that happens when integration is left at the end of a development phase, and possible integration problems arise earlier on in the project.

40-hour Week Developers must not work more than 40 hours a week. Overtime is accepted for one week, but never for two consecutive weeks. Having to work overtime is an indication of a problem for which working overtime is not the solution. Developers cannot produce the required quality when working overtime.

On-Site Customer A real customer representative should be physically present at the development site full-time. The customer can do other work at the time, but he must be available for the team to clarify user stories[2] and make business decisions.

Coding Standards Since all software code is being modified by everyone, the team must adopt a single coding practice that is accepted and followed by all team members. Developers should not be able to recognise who wrote which piece of code. Without an accepted standard development time will be lost on code style questions.

4.2.2 Common Problems

There are several problems identified when using XP. As noted before, McBreen [2003] surmises that developers who have used XP might not be willing to switch back to traditional, waterfall-type software methods. Another point is the scalability of the process. According to Maurer and Martel [2002], there is no real data available on how well XP scales for larger teams. Beck [2000] suggests using around ten developers in an XP team.

4.3 Scrum

Scrum is an agile, lightweight and adaptable process for managing systems development with small teams. Scrum does not define any specific development methods for the implementation phase, but concentrates on how the team members should function in order to produce the required system in a constantly changing business environment. [Scrum; Abrahamsson et al., 2002]

Scrum starts with the assumption that software development involves several variables that are likely to change during the project. These include requirements, resources, time frame and technology [Abrahamsson et al., 2002]. Instead of trying to plan these in

[2] User stories are XP's counterpart for requirements.

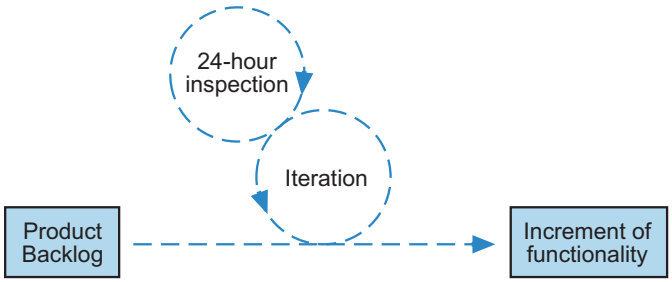

FIGURE 4.1: The Scrum Skeleton (adapted from [Mahnic and Drnovscek, 2005])

detail in advance, Scrum defines an empirical process control for ensuring that the current status of the project is visible, inspected and the methods are adapted to function in new situations [Mahnic and Drnovscek, 2005].

At the core of Scrum is an iterative, incremental process skeleton, which is shown in figure 4.1. Scrum is an iterative process: a project using Scrum is divided into *sprints*, which are the iterations of Scrum. Scrum is also incremental: each sprint produces a working, shippable version of the software with additional functionality.

The actual Scrum process is built on top of the skeleton. The process contains three phases: *pregame*, *development* and *post-game*. Before the project begins, the project teams, tools and other resources are defined in the pregame phase. The pregame phase also contains the high-level architecture definition of the system. The development phase is the agile part of Scrum, in which the software is implemented in sprints. All variables that can change (such as requirements and resources) are observed and controlled through various Scrum practices. After there are no more requirements for the software, the project moves to the post-game phase, in which the finalised software is documented and released. [Abrahamsson et al., 2002]

4.3.1 Roles

To implement the iterative, incremental skeleton shown in figure 4.1, Scrum defines three roles for project members [Schwaber, 2003; Mahnic and Drnovscek, 2005]. These roles are described below.

Product Owner The product owner represents the business viewpoint of the project by providing the initial overall requirements, objectives and release plans. The product owner is also responsible for prioritising the requirements throughout the project.

Team The team is responsible for developing the functionality in the sprints. The team is a self-organising unit who is solely responsible for producing the required function-

ality at the end of the iteration. The team itself selects the development methods used in the project and assigns the development tasks. A good size for a Scrum team is not more than ten people. The motivation for this is that small teams that work independently are more effective [Rising and Janoff, 2000].

ScrumMaster The ScrumMaster is Scrum's counterpart for a project manager. In contrast to a traditional project manager, a ScrumMaster does not lead the team, but rather acts as a coach. The ScrumMaster is responsible for maintaining the Scrum process and for teaching Scrum to everyone included.

A crude summary of the role distribution is that the product owner gives and prioritises the requirements, and all project members decide what should be developed in the next sprint. The team is then left alone to produce the required functionality, with the ScrumMaster supervising the correct usage of Scrum methods and acting as a mentor. In addition to these, Abrahamsson et al. [2002] identify the roles of *customer*, *management* and *user*. These roles are largely related to setting the goals and requirements.

Perhaps most importantly, Scrum divides the project stakeholders into those who are *committed* and those who are only *involved*. The product owner, the Scrum team and the ScrumMaster are all committed to the project, but another manager interested in the project would only be involved. In Scrum, only the committed people have a right to decide about project issues.

Schwaber [2003] reports of projects that had not worked as well as they could have when people had not filled the roles properly. For example, if the ScrumMaster tries to lead the team in the same way as a traditional project manager, the team does not achieve the commitment and productivity that can emerge when they are left to find out the best methods by themselves.

4.3.2 Practices

To control the chaos caused by unpredictability and complexity, Scrum defines a set of practices, which are listed below. The practices have been taken from [Abrahamsson et al., 2002; Schwaber, 2003; Rising and Janoff, 2000].

Product Backlog The product backlog of Scrum contains the prioritised requirements of the software to be developed. The product backlog is never completed during the project, but it is always updated when new requirements are discovered or current ones are refined. When all items in the backlog have been finished, the project is complete. The product owner is responsible for maintaining the product backlog.

A tool for maintaining the product backlog item status is the *burndown chart*,

which is a table that can be used for tracking the items. It shows the amount of work remaining for each item across iterations.

Effort Estimation Effort estimation in Scrum is an iterative process. At first, initial estimates are filled to the product backlog (or burndown chart). After more accurate estimations are formed, the new estimates are added as new columns to the burndown chart. This way the whole estimate history can be tracked from the burndown chart.

Sprint At the heart of Scrum are the sprints. A sprint is originally a 30-day iteration, but Rising and Janoff [2000] relax the length requirement into one to four weeks. Each sprint delivers valuable functionality to the developed system. During a sprint, the Scrum team selects the appropriate methods for reaching that goal, with the aid of the ScrumMaster.

Sprint Planning Meeting Before each sprint a two-part sprint planning meeting is held. In the first part of this meeting the goals for the next sprint are decided and prioritised. The product owner selects the most important features from the product backlog, and the team then tells how many of these they can implement during the sprint. In the second part, the team lays down preliminary plans for the sprint by creating tasks to the *spring backlog*.

Sprint Backlog The sprint backlog is the starting point for each sprint. Initially, a set of requirements is taken from the product backlog and expanded into the sprint backlog. These are then refined during the sprint, but new requirements are not added to the list. New requirements can only be added to the product backlog.

Daily Scrum Meeting To keep every project member up-to-date with the current situation, Scrum introduces daily Scrum meetings. These meetings are short status meetings that follow a very well-defined form. In each meeting, every developer is asked three questions: *what has the developer done since the last meeting*, *what does the developer plan to do before the next meeting*, and *are there any obstacles in the developer's way*. The purpose of these formal questions is to keep the meeting focused but still allow everyone to get a clear picture of how the project is proceeding. Any problems identified are not discussed in the meeting, but another meeting will be scheduled for the team members who should discuss the problem.

Rising and Janoff [2000] report that developers enjoy the small successes that arise from being able to say that they have finished their tasks. The meetings have also been useful as other developers have often battled with the same problems and could share their experiences after the daily meeting.

Sprint Review Meeting After each iteration, a sprint review meeting is held. In this meeting the produced software is introduced to the rest of the project stakeholders. In the meeting the contents of the next sprint are also discussed.

4.4 Adopting an Agile Process

When taking an agile approach into use, Hodgetts [2004] reports that trying to take all aspects of an agile methodology into use at once can lead to project failures. Not everyone is convinced at the start that the agile methods are the right choice; and without motivation, they will surely fail. Trying to work with many different new processes at once might not work even though the developers are committed – there's just too much to learn at once. The initial difficulties with agile methods might also be too much for the organisation to cope, if they are not introduced gradually.

Beck [2000] suggests that XP should be adopted one practice at a time, so that developers can become thoroughly familiar with the new practice and understand its purpose. The practice that is selected should be one that addresses the worst problem of the team.

4.5 Summary

In this chapter, an introduction to agile software development methods was given and two of the most popular agile methods have been presented. Section 4.1 described the agile viewpoint and explained what is different in agile methods, in comparison with the traditional software development methods.

Section 4.2 introduced Extreme Programming. XP has been the most popular agile method of late, and positive results have been reported when using it. However, especially the scalability of XP has been questioned.

In section 4.3, the Scrum method was presented. Scrum is another popular agile method which concentrates on the managerial aspects of software development, instead of providing any specific development practices. Scrum highlights the effects of social dynamics on productivity.

Finally, section 4.4 described how agile methods should be adopted. Instead of trying to tackle everything at once, teams should only take a few new practices into use at a time. Beck [2000] suggests identifying the most pressing problem and selecting only one XP practice to deal with it.

Chapter 5

Software Project Evaluation

In order to evaluate a software development project and its outcome, a set of evaluation metrics is required. The evaluation of a project is an important requirement for concrete software process improvement. An abundant amount of methods exist for measuring the software itself, but metrics for agile development processes are scarce. In this thesis, a metric for measuring the software itself is presented alongside with a metric for evaluating the adoption of Extreme Programming practices.

First, the Chidamber-Kemerer metrics [Chidamber and Kemerer, 1994] for measuring object-oriented software are presented. The CK metric suite is a known and used set of methods for measuring different quality aspects of software. After that, the Extreme Programming Evaluation Framework [Williams et al., 2004a] is introduced. The XP-EF is a set of metrics that provide information on how well the XP methods have been taken into use and how successful the team has been. The XP-EF metrics have been designed so that they can be used in a small team without a dedicated metrics specialist.

5.1 Chidamber-Kemerer Metrics

To improve the software process used in a company and the quality of the produced software, a method for measuring the software is required. With object-oriented software engineering, the traditional methods for evaluating software are no longer appropriate. Chidamber and Kemerer [1994] report that the conventional software metrics are lacking appropriate mathematical properties and are without a solid theoretical base. They proceed to propose a set of methods that are firmly rooted in theory and relevant to practitioners in organisations. After almost ten years, the CK metrics have been used in measuring software and Subramanyam and Krishnan [2003] present further evidence in support of a correlation between the values obtained from the CK suite and software defects.

The CK suite contains six metrics for measuring software design. The metrics are taken from [Chidamber and Kemerer, 1994] and they are summarised below.

1. **Weighted Methods per Class (**WMC**)**
   This metric is a weighted sum of all the methods in a class. In the original CK suite, a weight was assigned to each method based on the complexity of the method. The actual implementation was not specified, however, and some researchers suggest that the weight can be ignored, and the metric can simply count the amount of methods in the class [Subramanyam and Krishnan, 2003].

   The number of methods in a class predicts how much time and effort is required to develop and maintain the method. The method count has an impact also on the children of the class, since all but private methods are inherited to the children.

2. **Depth of Inheritance Tree (**DIT**)**
   This metric calculates the longest path from the measured class to a root class in the inheritance hierarchy. Longer inheritance paths indicate greater design complexity since more classes and methods are involved. Longer paths also allow for greater potential reuse of inherited methods.

3. **Number of Children (**NOC**)**
   This metric calculates the number of immediate children that have inherited from the measured class. Greater number of children indicates greater reuse, since inheritance is a form of reuse. However, a large number of children may also be caused by improper subclassing. In any case, classes with large child counts have a greater influence on the overall design of the system and thus require more testing.

4. **Coupling Between Object Classes (**CBO**)**
   This metric calculates the number of classes to which the measured class is coupled. The value is an indication on how independent the class is. Excessive coupling between classes is detrimental to modular design and prevents reuse. Highly coupled classes are also more sensitive to changes in other classes. Classes that have high coupling require more testing than independent ones.

5. **Response For a Class (**RFC**)**
   This metric calculates the amount of methods that can potentially be called in response to a message received by an instance of the measured class. This set is defined to be the union of all the methods in the class and all the methods that can be called from the methods of the class. Large values for this metric indicate that the class is complex and debugging it is more difficult since a greater level of un-

derstanding is required. More testing time should also be allocated for classes that have a large response value.

6. **Lack of Cohesion in Methods (**LCOM**)**
   This metric calculates the difference between the amount of non-similar method pairs and the amount of similar method pairs[1] in a class. Negative values are truncated to zero. Classes with more similar method pairs than non-similar ones are desirable, since they are cohesive. A large LCOM value indicates that the class contains separate functionality and should therefore be partitioned into multiple cohesive classes.

The CK metrics provide a suite of measurements that can be used to evaluate object-oriented software. The metrics can be used to track down key problem areas in a software system and provide a way to compare different projects.

5.2 XP Evaluation Framework

As discussed before, Extreme Programming has been gaining popularity, but empirical evidence concerning the use of XP has been scarce. One reason for this is the lack of proper metrics. One such metric is the Extreme Programming Evaluation Framework [Williams et al., 2004a]. The XP-EF comprises of three parts, namely, the XP *context factors* (XP-CF), the XP *adherence metrics* (XP-AM) and the XP *outcome measures* (XP-OM). Of these, the XP context factors are used to track down the exact context of the project, from software classification and sociological factors to ergonomic and geographic factors. These factors are important when considering the differences between two projects.

The XP adherence metrics are used to evaluate how well the project has adopted the XP practices. The evaluation of adherence is based on both subjective and objective measures. The subjective measures consist of a fifteen-part Shodan Adherence Survey questionnaire [Williams et al., 2004a], which is presented to each team member. The questions are all linked to single XP practices, and the answers form a subjective measurement on how well the practices have been adopted. All fifteen measurements can be combined to form a single value that indicates how well XP has been adopted on the whole. The measurements of the questions are weighted with the weight of the corresponding XP practice. The weight of a practice is defined to be the amount of other XP practices the specific practice supports. These weights are originally calculated from the XP practice support diagram presented in [Beck, 2000].

[1] Similarity of method pairs is the amount of same instance variables that the methods operate on; a similar method pair is therefore a method pair whose similarity is non-zero.

The objective measures consist of nine metrics that can be measured either automatically or manually. These contain metrics such as the ratio between test code and source code; and unit test runs per day. Both the subjective and objective metrics are listed in [Williams et al., 2004a].

The XP outcome measures is a set of measures that evaluate the produced software. The XP-OM includes calculating the CK metrics and McCabe Complexity[2] of the software, and furthermore requires the calculation of six more XP-specific metrics. These metrics are described in detail in [Williams et al., 2004a].

Although the XP Evaluation Framework is a relatively new framework, it has been already used in some projects. Williams et al. [2004b] report that the XP-EF has been comprehensive enough in a case study while not imposing too much burden on the team members. They acknowledge, however, that there is still work to be done in validating and extending the framework.

5.3 Summary

In this chapter, the need for measuring software systems and software projects has been identified. The chapter also presented two methods for these purposes.

Section 5.1 introduced the known and used CK metric suite. The suite has been used in measuring object-oriented software and has proven to be useful in identifying problem areas in software systems.

In section 5.2, a relatively new framework for evaluating Extreme Programming was presented. The XP Evaluation Framework consists of several subjective and objective metrics that can be easily calculated alongside a project. The metrics provide information on how well the XP practices have been adopted and how high the quality of the produced software is.

In conclusion, it is important to realise that to gain the most value out of any software metrics, a comparison point should be available. This can be another project, or an earlier version of the same project. The metric values can still be used by themselves to locate problem areas in software, or to obtain subjective evaluations of the project.

[2] An introduction to the McCabe Complexity can be found, for example, in [VanDoren, 1997].

Chapter 6

Project: KAPSELI

This chapter describes the KAPSELI-project[1] and its requirements. In the project, HiQ Softplan Oy creates a new implementation of a web-based real estate transaction system for GVA Finland Oy. Crucial attributes of the new system are extensibility, maintainability and modularity. The open-source solutions and agile software development methods presented in this thesis are applied in the execution of the project.

First, the current situation and future needs of HiQ Softplan Oy's client, GVA Finland Oy, are presented. The current system is described and the need for a new system is identified. After that, the requirements for the new software system are summarised.

6.1 Current Situation

GVA Finland Oy uses an existing web-based real estate transaction system. The administrative side of the system is used to record information about facilities that are on sale or for rent and customers that are selling, renting or purchasing these facilities. Other companies can use the public side of the system to search for facilities for their own use. Both the administrative side and the public side are accessible with a standard web browser.

The current system works fine for this purpose, but GVA Finland Oy has new requirements and the system needs to be extended. Furthermore, an interactive map service needs to be integrated into the system, and the customer information needs to be moved into a customer relationship management (CRM) system. In addition, support for generating reports needs to be included in the system.

The new business requirements set more demands for the architecture of the system. It was decided to reimplement the system because of the new demands and because the original source codes were not available at first. The new system is designed to have a more robust structure and backing frameworks. The software will be implemented on

[1] KAPSELI is an abbreviation of the Finnish words *kiinteistöt, asiakkaat, projektit, seuranta, lisämyynti*.

top of widely used open-source solutions using agile software development methods.

6.2 Software Requirements

The basic requirements for the first release of the new real estate transaction system are threefold. First of all, the new system must contain the same functionality than the existing system with the exception of customer management, which is moved to a CRM system. Secondly, an interactive map software, which will be provided by a third party, needs to be integrated to the system. Finally, the customer management side needs to be integrated to a CRM system provided by a third party.

In addition, the software architecture needs to be implemented in such a way that it supports extending the basic functionality later on. Future extension ideas include custom reports and different types of sales projects. Some changes were also planned for the business object representation and handling in the administrative side of the system.

It was also decided to renew the layout of the system. In addition to creating a new graphical layout to the system, some changes were planned for the administrative forms used to manage the real estates in the system. The look and feel of these forms was not finalised during the planning phase of the project, and it was decided that these will be iterated during the development process.

Chapter 7

Selected Approach

This chapter presents the approach selected for carrying out the project described in chapter 6. First, the selected open-source solutions used in the project are listed. Secondly, an overview of the the architecture designed for the new system is presented. Thirdly, the agile methodologies that were chosen for the project are given. After that, the project team is introduced. Finally, the methods used for evaluating the project are presented.

7.1 Selected Open-Source Solutions

To minimise the amount of code needed to implement in-house, all the open-source solutions introduced in chapter 3 were selected for use in the project. The solutions and their usage in this project are listed below.

Apache Struts Struts was selected as the web framework to be used in the project. In the new system, Struts is responsible for managing the dynamic web page creation and processing the input sent by the user. JavaServer Pages will be used as the view components that will render the actual XHTML content.

Spring Framework Spring was selected as the application context that creates and configures most of the objects used in the system. Although Struts has its own way of creating the required action objects, Spring and Struts can and will be integrated so that the Struts action objects are under Spring control.

Hibernate Hibernate was selected to handle data persisting to database. All persisted objects in the system are annotated with special Hibernate metadata, so that they can be automatically stored to and retrieved from the database. Spring is configured to select Hibernate as the data source for the application.

XDoclet XDoclet was selected for automatic artifact generation. In this project, XDoclet

is configured to automatically generate Hibernate mapping data and parts of Struts and Spring configuration files from XDoclet tags written in Java source code files.

Eclipse Eclipse was selected to be used for developing the system. The source code supporting tools in Eclipse's Java Development Toolkit have been found useful in previous projects.

Apache Maven Maven was selected to handle the building of the system. Especially the idea of not having to specify the project paths and Java classpaths was seen as an appealing idea.

Subversion Subversion was chosen as the version control system for the project. Subversion was selected over the tradition CVS since it has been developed as a replacement for CVS and it addresses some of the shortcomings of CVS.

7.2 System Architecture Overview

Since the current system is a standard web application that is used with a normal web browser, it was decided that the new system will also be a normal web application. The three-tier model, which was described in section 2.4, was selected as the basis for the implementation.

The most important components of the designed architecture are shown in figure 7.1. The lines connecting the components describe the response handling path of a standard query initiated from the user's browser. The numbers in the figure correspond to the steps described below.

1. User initiates a request by clicking on a link or by posting a form in the system. The request is received by the Struts `FrontController`.

2. The `FrontController` generates or populates the Struts `ActionForm` associated with the request from the request data. Query parameters are automatically parsed and inserted into the form.

3. The `FrontController` selects the proper application-specific action component, and passes control to it. The prefilled `ActionForm` is passed to the action as a parameter.

4. The action loads required data from data access objects, which are implemented using Hibernate.

5. The data access objects automatically load the required business objects from the database, and return them to the action component.

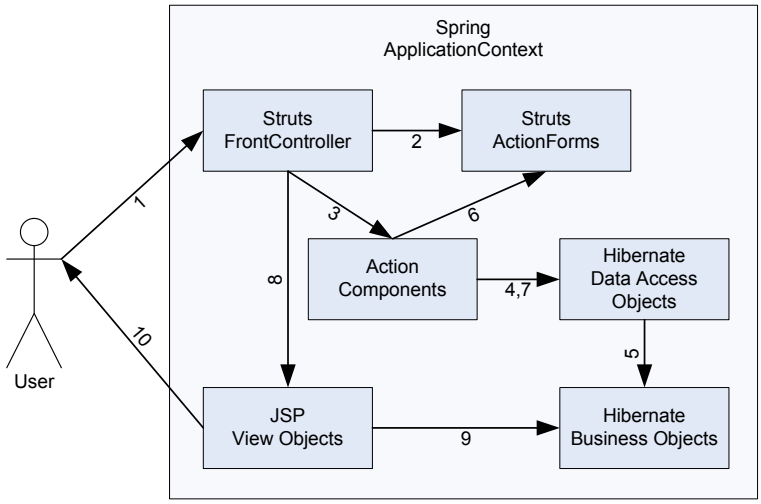

FIGURE 7.1: System Architecture

6. The action component performs the required actions on the business objects as requested by the request parameters and `ActionForms`. Data from `ActionForms` is transferred to the business objects.

7. The action component persists the modified business objects by using the data access objects, and returns control to the Struts `FrontController`.

8. The `FrontController` selects the proper view to show to the user based on the return value of the action component. The action component can indicate whether the action was processed successfully, or if an error page should be shown.

9. The selected JSP page is rendered by using the retrieved business objects as data sources. The design was simplified by allowing the business objects to work as data transfer objects (and as helper objects in the front controller design pattern).

10. The rendered page is sent back to the user's browser.

The architecture for the web components is largely dictated by Struts. Struts implements the model-view-controller design pattern with the `FrontController` and actions working as the controller, the JSP pages as the view, and the application-specific business objects and data access objects as the model. The front controller design pattern is also realised with the `FrontController` provided by Struts working as the controller and dispatcher, the JSP pages as the view, and the business objects as the helpers.

The business objects in the system are initialised using the Spring application context. Spring configures the objects using the inversion of control design pattern. The map-

pings between the objects are written to a configuration file, and Spring is responsible for setting up the links between the objects via setter injection.

The database mappings in the database tier are configured with the object/relational mapping provided by Hibernate. Special Hibernate metadata is annotated in the business object Java code files.

7.3 Selected Agile Development Methods

This section lists the agile software development methods selected for use in the project, and presents the criteria for selecting them. For the project, methods from both Extreme Programming and Scrum were selected. As suggested in section 4.4, it is better to only select a few methods for use instead of trying to tackle all the methods given in an agile methodology at once.

7.3.1 XP Practices

For developing the software, a subset of six XP practices was selected. The selected practices are listed below alongside with justification for their selection.

Short Releases Without short releases the design for the system needs to be finalised before the start of the implementation phase. With short releases, it is possible to leave some implementation details a bit unclear at the beginning of the project and iterate them with the customer between the releases.

Refactoring Since some of the requirements are likely to change somewhat during the implementation phase, the code needs to be refactored from time to time.

Testing Testing is required to make refactoring possible. Testing was also selected because it has been seen to produce better quality, as discussed in section 4.2.1, and because it has been identified as the most important XP practice.

Collective Ownership Collective code ownership is another requirement for refactoring. To be able to refactor efficiently, developers need to be able to modify every part of the codebase.

Simple Design Since refactoring is expected to happen frequently, the design should be kept simple. Simple designs are more easily changed and are less error-prone.

Coding Standards Coding standards were also chosen to support refactoring. For this project, coding standards mean indentation rules as well as rules about where different information should be placed.

This subset of XP was seen to be small enough to be easy to adapt to. The selected practices were chosen so that they support each other as much as possible. For example, the subset would not be functional if refactoring or testing was left out.

7.3.2 Scrum Practices

Although some authors suggest adopting agile methods only a few at a time, they are most often talking about XP practices or other development practices. When talking about Scrum, it is suggested that Scrum should be adopted all at once. However, it was decided that the Scrum review, meeting and the practices between sprints will be left out because the project is small. The estimated size for the project is only a couple of sprints. The selected practices are listed below with justifications for their selection.

Product Backlog The product backlog was selected to function as the requirement repository, which contains all the currently known tasks. The backlog will be updated whenever a new requirement is received or when additional information for an existing requirement is available.

Effort Estimation The client required an approximate effort estimation for the entire project before the start of the implementation phase. The initial estimates were selected as the starting point for the project effort estimates, even though this does not exactly adhere to Scrum ideology. In pure Scrum, only the requirements for the next sprint should be estimated.

Sprint To support the XP short releases practice, the Scrum sprints were selected as the short iterations.

Daily Scrum Meeting The daily Scrum meetings were selected as the main team organisational activity. Since both the project and the project team is small, this meeting was seen as an adequate way of keeping everyone up-to-date with the current project status.

For this small project, this adoption of the Scrum process was seen as an adequate set of project management practices. The practices were selected so that they support the XP practices selected in section 7.3.1.

7.4 Project Team

The development team for the project consists of three developers from HiQ Softplan Oy. Two of them are currently writing their Master's Theses, and one is a graduated Master

of Science. All three have over five years of work experience behind them in the software engineering field.

The team members possess a good knowledge of Java frameworks, but they are not intimately familiar with all of the selected solutions. Furthermore, the team members have not been working on a project that uses agile methodologies to any greater extent before, but have studied the theories behind the methodologies.

7.5 Evaluation of the Project

To evaluate the outcome of the project, some concrete measurements need to be made. This section lists the methods selected to evaluate both the outcome of the project and the project itself. To achieve this, the CK metrics were selected to measure the developed software, and the XP Evaluation Framework was selected to evaluate the adoption of the XP practices.

7.5.1 Software Quality

The Chidamber-Kemerer metrics suite (see section 5.1) was selected for measuring the quality of the produced software. All the six metrics in the CK suite will be measured with the open-source tool AOPMetrics [AOPMetrics] at the end of the project. AOPMetrics supports measuring both AspectJ and normal Java code. Thus the metrics used in it are named a bit differently than the standard CK metrics, but they are equivalent. As AOPMetrics is a free open-source tool, it was selected for use in this project.

The problem with measurements is that they are not useful by themselves, but require a comparison point to provide concrete feedback. Unfortunately the source codes for the original software are not available. However, Chidamber and Kemerer [1994] and Subramanyam and Krishnan [2003] provide a total of four measurements of different software systems. These shall be used as a comparison point in evaluating the obtained CK metric values.

7.5.2 Agile Method Adoption

The XP Evaluation Framework (see section 5.2) will be used to evaluate the adoption of agile practices in the project. Of the three parts of XP-EF, only the XP adherence metrics will be measured. In addition, only the practices that were taken into use will be evaluated. This was selected as the approach because the XP context factors are mainly useful for comparison purposes between multiple projects and the XP outcome measures contain very few other metrics than the CK metrics that could be measured with the

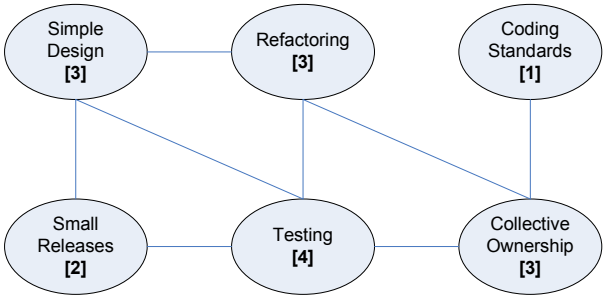

FIGURE 7.2: XP Practice Weights

selected XP practices.

The questionnaire used for the XP adherence metrics subjective measures is shown in appendix A. The questions will be asked from all team members, and the average percentages derived from the answers will be used to subjectively measure the adoption of the corresponding practices. The weights shown in figure 7.2 are used to obtain a weighted average of the answers. This value tells the overall adoption of XP practices. The figure was adopted from [Beck, 2000] by removing the practices that were not taken into use in this project. The weights of the practices were obtained by calculating the amount of other practices a single practice is connected to. In addition, the objective measures listed in [Williams et al., 2004a] will be measured for the practices that were taken into use. The selected metrics are shown in appendix B.

7.6 Summary

This chapter explained the approach selected to execute the project described in chapter 6. The selected solutions and methods were presented.

Section 7.1 listed the open-source solutions selected for use in the project. All the solutions presented in section 3.3 were selected for use in the project. The solutions were selected to both cover the architecture of the system and to provide support to the software development process.

In section 7.2, an overview of the system architecture was given. The system architecture was largely dictated by the selected open-source frameworks, which is often the case. Because the developed application is a web application, Apache Struts defines the basic structure of the architecture.

After that, section 7.3 described the agile practices selected for use in the project. A subset of both XP and Scrum practices was adopted for the project. The practices were selected so that they support each other as much as possible.

In section 7.4, the project team was introduced. The team consists of three experienced software developers, ready for their first agile project.

Finally, section 7.5 presented the evaluation methods that will be used to measure and evaluate the project. The CK metric suite was selected to measure the produced software, and the XP adherence metrics of the XP Evaluation Framework were selected to evaluate the adoption of XP practices.

Chapter 8

Evaluation and Analysis of the Project

This chapter evaluates the produced software and the processes used in the project. The first section begins with an evaluation of the suitability of the selected open-source solutions in agile software development. After that, the produced software is evaluated based on the measured CK metric values. The second section describes the agile software development methods selected for the project. First, the adoption of XP practices is evaluated by using the XP Evaluation Framework. The section ends with an analysis of the usage of both XP and Scrum in the project. Finally, a high-level summary on the outcome of the project is given.

8.1 Software

A comprehensive set of open-source solutions was selected in section 7.1 to work as a basis for the implementation and to support the development process. This section begins with a subjective analysis of how well the selected software performed in the agile development process. After that, the software quality is evaluated and analysed based on the measured CK metric suite values.

8.1.1 Open-Source Solutions

The frequent refactorings that were applied set high expectations especially on the maintainability of the open-source software used in the project. The evaluations for the selected solutions are given below.

Apache Struts Struts was used as the core of the web components of the system. In the implemented system, the web components form the largest part of the system.

This means that most of the application architecture was dictated by Struts. This can be seen to contain both positive and negative aspects. Because the Struts architecture has been developed over a period of many years, and it has been used in a multitude of projects world-wide, the architecture can be expected to be more robust and extensible than what could have been produced in-house for this relatively short project. The negative side of this is that the development team did not have as much freedom when designing the system. However, the team did not encounter many problems caused by this.

In other areas there were some inconveniences, especially with the integration to Spring. For example, creating a new Action component required writing configuration information to a total of three different configuration files (the configuration files of Struts, Spring and Tiles[1]). This made the maintaining of the configuration files laborious and error-prone. However, there were not as many Struts Action configuration changes as feared, even though the system code was refactored often. Because of this, the maintenance of Struts configuration was not such big of an issue. Nonetheless, if and when structural changes need to be applied to that configuration, some problems can be expected. Another shortcoming of Struts was that there was no simple way of applying multiple actions to a single request. Because of that, authorisation checking had to be done in all action components instead of configuring it in the Struts configuration file. The next major release of Struts introduces action chaining which might provide a better solution for this issue.

Another problematic area with Struts was unit testing. Plain Struts does not offer much in way of supporting unit testing. In the end, unit testing the action components was accomplished by creating dummy request and mapping objects and executing the actions by hand. However, this approach did not allow testing of the view components.

The problems noticed were not critical, however. On the whole Struts worked fine and provided a robust basis for the web components of the system. In addition, Struts 2, which is the next major release of Struts, promises to address some of the issues the current version of Struts has.

Spring Framework Spring delivered exactly what it promised to deliver to the project. There were no major problems with the creation and configuration of the system objects. The dependency injection mechanism proved to be a convenient way of setting up the relationships between system objects. This allowed rapid develop-

[1]Tiles is a template engine for creating a common look and feel for web applications. Tiles is shipped as a part of Struts, but it can be used without Struts too. In this project Tiles was integrated to Struts.

ment of new objects. However, since all the relationships of the objects were set from outside of them, it was not apparent from the code if all the required dependencies were properly initialised. In some occasions, a link was forgotten between two objects, which was not seen before the software crashed when trying to use the nonexistent link. Spring does provide the means for calling an initialisation method after the dependencies are set, which makes it possible to check that the object has been fully initialised. Nevertheless, it is still easy to forget either the method itself or the configuration required to call the method. Another solution for this would be to use constructor injection, in which the constructor can check that all the required links are set. However, with this approach, creating circular references[2] is not possible.

Using Spring turned out to be helpful for unit testing, because every system object was available for the unit tests via the Spring application context. The ease of creating and configuring new objects was also beneficial for refactoring. All in all, Spring performed rather well in this project.

Hibernate Managing the business object mappings to database proved to be easy with Hibernate Annotations[3]. The basic approach of keeping the mappings alongside with the objects made rapid development possible. In addition, Hibernate can be configured to automatically create and update the required database table structure. In effect, no SQL table creating scripts were required for the project. Furthermore, all the database queries could be created by a query criteria creation facility provided by Hibernate. Because of this, the project contains no hand-written HQL (or SQL) queries. The database-related code is cleaner and easier to maintain with the object-oriented query generation functions.

However, some of these benefits have their downsides as well. Annotating the business objects ties the objects themselves to the database tier. For this project this was not seen to be such a big problem, but for larger objects a separate interface or implementation should perhaps be created to keep the different application tiers properly separated. Another problem was transaction management. In this case, the problem was not in the transaction management system itself, but rather in how many ways there are to manage the transactions. Hibernate and Spring provide numerous different ways in which transactions can be configured, such as manually coding the transactions and automatically setting up the transactions with annotations or configuring them with aspects. In addition, manual transactions can be

[2] A circular reference is a situation where the dependency path of a set of objects is not a tree.

[3] Hibernate Annotations was used in the project instead of configuration files generated by XDoclet; see the comments on XDoclet for explanation.

configured either with Hibernate itself or with wrappers provided by Spring. The proper transaction management strategy selection was further complicated by automatic transaction creation for incoming requests to Struts actions.

Testing Hibernate was easy with Hibernate–Spring integration. Database connections and connection pooling could be managed via the Spring configuration file, and data access objects could be fetched via the Spring application context. Refactoring the data objects was also easy, because the annotation could be updated at the same time. The only downside was that if the type of a property was changed, Hibernate did not update the database table column accordingly. Also, Hibernate does not delete old columns or tables from database, so some manual work still had to be done.

XDoclet This was clearly the most problematic tool selected for the project. Originally, XDoclet was supposed to create the Hibernate mapping files, and Struts and Spring configuration files. Shortly after the project started, Hibernate Annotations was released for production use, and the separate mapping files were replaced with annotations. Generating the Struts configuration files worked fine, except for merging the static and dynamic configuration parts, which did not seem to work at all. In effect, the configuration file had to be manually merged from the generated file and the static files. Spring configuration file generation did not work at all. In the end, it was decided to leave XDoclet out of the project.

It is possible that the problems experienced with XDoclet are only specific to XDoclet 2. It seems that the project might be dying out, because there have been no releases or site updates for a year and there is virtually no documentation at all available for the tool.

Eclipse As expected, Eclipse supported the agile development process well. The refactoring tools were extremely useful when applying structural changes. Especially the automatic renaming tools proved to be an invaluable aid in refactoring. Another useful tool was the problem list, which provides a list of current compilation failures and warnings with links to the offending locations. In the simplest case, it was possible just to change some functionality (for example, method parameters), and then fix all invocations of the method by navigating via the problem list.

Apache Maven Maven was selected for the project because it promises to simplify the build process. The standard directory structure turned out to be extensive, and the building of the project was effortless, after Maven had been properly configured. However, configuring Maven for the first time was a tedious task. This was partly due to poor or missing documentation. In some cases, the documentations

of Maven 1 and Maven 2 were intermingled, even though the functionality had changed between the releases. In addition, configuring the Maven repository for libraries that were not available in the Maven central repository presented some problems at first.

Another problematic area in Maven is custom tasks. The idea of the build lifecycle is integrated strongly in Maven, and executing tasks outside the normal lifecycle is not directly supported. To create a custom Maven task, a Java plug-in has to be created. Maven also supports running custom Ant tasks with the Maven Antrun plug-in. However, there seems to be no way of creating different Ant tasks that could be run separately, because the tasks run outside the lifecycle are selected by the plug-in name.

Overall, Maven was perhaps too complex for this project. On the other hand, Maven was mainly used as the build and dependency management tool. The reporting and documentation features of Maven were not used. Because of this, the additional value of Maven over a traditional build tool was not self-evident.

Subversion There were no problems encountered with using Subversion. In fact, because Subversion integrates with the team development tools of Eclipse, the usage of Subversion is identical to using CVS. It is expected that most of the benefits of Subversion over CVS will become apparent later on, because the modification histories are available even if a file has been renamed or moved under a different directory.

There is one problem with Subversion's preference of using HTTP as the transport protocol, namely, proxies that do not support all the required commands. There are some HTTP proxies that cannot be used when connecting to Subversion repositories. For these cases, the stand-alone Subversion server can be used, but it is not as straight-forward to configure, at least if the Apache version needs to be running too. However, if proxies are not required, there does not seem to be any reason not to use Subversion over CVS.

As a summary, most of the solutions selected worked well in the project. The tools that did not perform as well were not mentioned as often in literature as the tools that did. This supports the claim that using mainstream open-source software is a lasting solution. The tools and frameworks that had a strong community and long use history worked well and were updated frequently, even during the execution of this project.

8.1.2 Software Quality

The produced software quality is evaluated and analysed in this section. Basic software size metrics are given in table 8.1 as background information. The lines of class code shown in the table represent the lines of executable program statements, with comments and blank lines omitted.

TABLE 8.1: Software Size Metrics

Metric	**Value**
Number of Classes	184
Number of Production Classes	120
Number of Test Classes	64
Lines of Class Code (LOCC)	11679
Production LOCC	6545
Test LOCC	5134

The metrics obtained from the CK metric suite are presented in table 8.2. The values were obtained using AOPMetrics [AOPMetrics]. Only the values for the actual production code were calculated, since the metrics for the test code were not considered meaningful. Because some of the names used in the CK suite differ from the terms used in AOPMetrics, both of the terms are listed in the table for the metrics with differing names. In these cases, the CK metric suite name precedes the name used in AOPMetrics.

TABLE 8.2: CK Metric Values

Metric	**Mean**	**Std.Dev.**	**Median**	**Min**	**Max**
Weighted Methods per Class Weighted Operations in Module	8.32	11.60	4	0	74
Depth of Inheritance Tree	1.25	1.33	1	0	4
Number of Children	0.34	1.76	0	0	18
Coupling Between Object Classes Coupling Between Modules	3.58	4.15	2	0	23
Response For a Class Response For a Module	14.02	16.71	8	0	85
Lack of Cohesion in Methods Lack of Cohesion in Operations	69.99	272.80	0	0	2246

To evaluate the measured values, the four software systems analysed by [Chidamber and Kemerer, 1994] and [Subramanyam and Krishnan, 2003] shall be used as comparison points. These systems are named A, B, C and D for the purposes of this comparison. Basic characteristics of these systems and the metric values are presented in appendix C. Because the systems are different and the similarity of the measurements cannot be verified, no hard conclusions can be drawn from the comparisons. However, the comparisons should give a subjective viewpoint of the general quality of the produced system. The comparison and evaluation results are listed below.

Weighted Methods per Class The WMC metric values of the developed system are similar, but somewhat below the values in all of the four comparison systems. Since the WMC metric is a simple prediction for the time and effort required to maintain the class, the values obtained point to the conclusion that the developed system should not be overly hard to maintain.

Depth of Inheritance Tree Again, the DIT metric values are equal or somewhat below the values of the compared systems. These values indicate that there are no overly long hierarchy paths that would complicate system maintenance. The values obtained from this metric support the conclusions derived from the WMC metric values.

Number of Children The NOC metric values have only been calculated for systems A and B. The values measured from the produced system are again lower than in the these systems, whose average maximum value is 46. This can be explained simply with the fact that systems A and B are much larger than the system developed in this project.

Coupling Between Object Classes The comparison systems had some variance in the values for the CBO metric. Systems A and B had relatively large maximum values, but the median value for system A was zero. Compared to the other systems, the median and mean values are on the same scale, and the maximum value of the developed system is lower. Thus, the developed system should not have excessive couplings.

Response for a Class The RFC metric values were not calculated for systems C and D. System A has approximately same median value but a larger maximum value, and the RFC values for system B are several times larger than the values for the developed system. According to this metric, the classes should not be too complicated. This conclusion is in line with the results from the CBO metric.

Lack of Cohesion in Methods Again, the LCOM metric has only been measured for systems A and B. However, the values in those systems are far lower than the maximum value for the measured system. The maximum values for systems A and B are only 200 and 17, respectively. However, the large values in the developed system are caused by a group of classes with very high LCOM readings. There are twelve classes in the system with an LCOM value that is over 100. In contrast, there are 68 classes whose LCOM value is zero, and a total of 85 with an LCOM value that is under ten. The twelve offending classes are Struts `ActionForms`, business objects, and search criteria classes. A connecting property for all these classes is that they have multiple property fields that have accessors and mutators (getters and setters). The problem with the LCOM metric is that, by definition, accessors and mutators increase the LCOM value, since they access only single properties and thus are not cohesive. The suggestion for large LCOM readings is to refactor the classes into smaller pieces, but that does not seem to be reasonable for classes that model domain information or are used to transfer data. To get more realistic readings from this metric, perhaps it would be better to leave out accessors and mutators from the calculations.

As a summary, the metrics seem to indicate that the quality of the produced system is at an acceptable level, even scoring somewhat better results for some metrics in comparison to the four other systems. The only exception is the value of the LCOM metric, which is excessively high. However, as discussed above, this is probably caused by classes that are not suited for the metric instead of bad design as such. According to the metrics, the maintenance of the developed system should not present any special problems.

8.2 Agile Practices

This section presents the evaluations of the selected agile methods and their adoption. First, the XP practice adoption is evaluated using the XP Evaluation Framework. After that, subjective comments on the adoption of Scrum are given.

8.2.1 Extreme Programming

The XP Evaluation Framework [Williams et al., 2004a] introduces a triangulated XP adherence analysis, which was adopted for this work and presented in table 8.3. The *adherence hurdle* and *comment* columns have been merged into a single *comment* column. The table contains the values from the Shodan Input Metric Survey (appendix A).

TABLE 8.3: Triangulated XP Adherence Analysis
DOA stands for Degree of Adoption, PW for Practice Weight.

Practice	DoA	PW	Comment
Testing[4]	50%	4	Unit tests for most of the functionality was created, but not for all. Schedule pressure and working habits prevented full adoption.
Refactoring	67%	3	Some team members did a lot of refactoring, others refactored only when refactoring was directly connected to the work at hand.
Short Releases	77%	2	The project has only one major release, but several sprint releases were set up on a test server for the customer to test.
Coding Standards	90%	1	Code formatting rules were configured to Eclipse's code formatter. Coding and logging practices were mostly obeyed directly, and the few divergent cases were quickly refactored to match the standards.
Collective Ownership	90%	3	Team members had no problems in changing code written by other developers.
Simple Design	77%	3	In a few cases some functionality that was known to be included later on was taken into consideration even though it was not directly used.

The results indicate that the selected XP practices were mostly adopted satisfactorily, with an overall adoption degree of 72%[5]. However, there were some practices that were not adopted as well as they could have been. A subjective analysis of the results and the experiences from the project follows.

Testing Testing was problematic, because writing unit tests for every possible action of the system was not seen to be fully justified, although the team was aware of the benefits from having an extensive test suite. Furthermore, because the team was

[4]Testing contains the average values from both automated unit tests and test first design.
[5]This value is the weighted average of the individual practice adoption degrees.

often in a hurry to make a feature work, not all of the tests were actually written before the code, but were supplied after the functionality was written. One problem was that because unit tests do require the skeletons of the final classes to be written before the tests, it was too easy just to create the full functionality while creating the skeleton. However, it was identified that even tests written afterwards are better than no tests at all.

Short Releases The team produced some intermediate releases for the customer to test. However, there was no schedule to the releases and the releases were given out whenever some new functionality was added to the system. This worked, because the team was small, code was continuously integrated to the version control system and deploying a new release only took a quarter of an hour. For a larger project an approach like this would not work, and the releases would have to be scheduled beforehand.

Coding Standards The coding standards were adopted with surprisingly few problems. The developers could agree on the code formatting rules as well as on the standard practices for logging and information placement. Again, the smallness of the team was helpful in agreeing on the common standards.

Collective Ownership This practice was also adopted with no problems. Especially the coding standards were helpful in the adoption of collective ownership. It seems that without good coding standards this practice cannot function very well, as developers could potentially start changing an existing component to suite their immediate needs without regard to the intended purpose of the component. This could be a serious problem in a larger project.

Simple Design In most cases, the simple design practice was adopted quite well. However, in a few cases the design was taken further than what would have been absolutely necessary for the situation. In these cases the requirements were known to be included in the final release, so the design beforehand was considered a safe bet. Another feeling the team had was that for core components of a large system the simple design might not be the best approach, because massive refactorings would be required later on. In these cases it might be better to design a bit further to keep the amount of refactoring needed low.

In addition to the subjective metrics, the objective metrics of the XP adherence metrics were also measured for the relevant practices. The results of the measurements are listed in table 8.4. According to the results, the unit test suite was run quite often. This is explained simply because running the test suite only took around a minute, so it could

be easily run often. Test code ratio was still relatively high, even though not all functionality was tested. However, the test suite has proven very useful in quickly determining whether a larger change has caused more than the intended effects. Release lengths have also been kept quite close to what XP proposes, even though they were not scheduled.

TABLE 8.4: Objective XP Adherence Metric Values

Metric	**Practice**	**Value**
Unit Test Runs / Person Day	Testing	3–4
Test LOC / Source LOC	Testing	78%
Release Length	Short Release	Release: 3 months Iteration: 2 weeks

As a summary, the selected subset of XP has been adopted mostly successfully with only a few setbacks. One key factor in this is perhaps the fact that only a few of the practices were taken into use at once, so there were not too many new practices to tackle. Another one is that the team was so small. For a larger team, the practices would need to be followed more strictly, and more resources should be assigned for their implementation.

8.2.2 Scrum

In contrast to XP, Scrum was not adopted equally well. This was perhaps due to the fact that the project team and the project itself was so small that there was no need for strict project management practices. Another point is that some of the XP practices also served in coordinating the team's work. The selected Scrum practices and their adoption are commented below.

Product Backlog A backlog was adopted and kept posted on the wall for everyone to see. In addition to the list, the team had three columns arranged on the wall, where post-it notes could be posted. Items taken from the backlog were moved from the *todo* column to the *active* column and finally to the *done* column. The Scrum-Master kept the status of the items on the backlog up to date, but not all of the requirements were added to the backlog itself. Some of them were just posted on the columns. In a few small cases, no tracked item was even created. This worked well for this small project, but for a larger project, the requirement tracking needs to be done more rigorously.

Effort Estimation All of the original requirements were estimated at the beginning of the project as the customer required them. However, when new requirements were added or previous requirements were expanded, new estimations were not always made. In addition, because there were no predefined schedules for the intermediate releases, the requirements for a given sprint did not have to be estimated very accurately. The team could work with the original, imprecise estimates as well.

Sprint The sprints adopted for the project turned out to be somewhat informal. There were no precise schedules and only a few planning meetings. The requirements for each sprint were not clearly defined, but instead a developer worked on a component for as long as it took to finish it, and then moved to the next component. Since the team was small enough, the ScrumMaster could still have an overview of the current situation and be assured that the project was heading the right way.

Daily Scrum Meeting The daily Scrum meetings were kept most of the days. However, the team did not have an exact time specified for the meeting, and the meeting was held whenever all the team members happened to be at work. This soon turned out to be a bad idea, since the meeting invariably interrupted with the developers' coding flow. Nonetheless, the meetings were seen to be an effective way of keeping everyone up to date, although the situation was quite clear anyway because the team members worked around the same table.

As a summary, the team did not strictly adhere to the selected project management practices, because both the team and the project was so small, and there was no pressing need for rigorous project management. However, some of the practices were found to be useful even if they were adopted only partly. For any larger project, even for a single ten-man Scrum team, the practices need to be followed more thoroughly.

8.3 Summary

This chapter described the suitability of the open-source solutions in agile development, presented the software quality metric values and analysed the adoption of the agile practices. Section 8.1 presented the selected open-source solutions and analysed their usage in the project. Most of the selected software worked well, and provided a robust basis on which the software could be built. However, some problem areas were also identified when using less widely-used open-source solutions. Most pressing of these is the lack of proper documentation; one tool was practically undocumented. The findings of the project adhere to the findings presented in literature; namely, using mainstream open-source software is a safe solution, but less well-known OS solutions do have their

risks. Afterwards, the CK software quality metric values were presented and analysed. According to the values, the produced software is of good quality, and maintaining it should not present any special problems.

The agile development methods selected for the project were analysed in section 8.2. The adoption of XP was evaluated using the XP Evaluation Framework. The values from the adherence metrics indicate that the selected XP practices were mostly obeyed in the project, but there is space for some improvement in the practice adoption. After that, some subjective comments on both XP and Scrum were given. Especially Scrum was not followed through rigorously. However, both the team size and the project itself was small, and there was no need for rigid process adherence. In the end, the team managed to finish the project by using a very lightweight approach. Especially the XP development practices worked well in this small project, allowing the team to design the software for only the current needs.

Chapter 9

Conclusions

In this thesis, the current state of web software development was analysed and its requirements were identified. Consequently, software solutions from the open source market were presented to meet those requirements. Furthermore, the relatively new agile methodologies were introduced as an alternative approach to software development. Finally, a suite of mature, *de facto* open-source solutions was selected for use in a real business software project that was managed using a subset of Scrum and XP agile software development practices. The produced software was measured using a set of known software metrics, and the agile development practices were evaluated with a framework designed for that purpose.

9.1 Produced Software

The software created in the project was developed using the selected XP development practices. The basic structure of the application was largely dictated by the Apache Struts framework, which was selected as the web framework for the project. In addition, the other open-source solutions used in the project also set some limitations on the software architecture. However, these limitations were not necessarily a problem. By using mature frameworks, the developed software inherited a robust, well-tested architecture that is more extensible and maintainable than what could have been created in-house with the available resources.

For the development work, the team followed XP's simple design practice, and the software was designed only to meet the current needs. Consequently, constant refactoring was required to maintain software quality and to meet new requirements. To support this, the team maintained an extensive unit test suite. The developers could be convinced that a structural change to the software had not broken anything by accident. Finally, the completed software was evaluated by using the CK metrics suite, which contains six

metrics that measure object-oriented design complexity. The resulting values were compared to the values of four larger software systems, whose metric values were presented in literature. Based on the comparison, the software design is not too complicated and maintaining the software should not present any larger problems.

9.2 Open-Source Solutions

On the basis of the results of the project it can be concluded that the open source community has strong solutions to offer for the web software development industry. There should be no reason why this finding would not be true for other fields of software engineering as well. Mature OS solutions that have a strong user community behind them have proven to be robust, extensive and efficient. The best OS solutions are lacking none of the features provided by their commercial counterparts. In addition, with open-source software, a company always has the option of customising the software to its own needs, should it be necessary. However, the OS solutions that are not as widely used do contain elements of risk. It is possible that a small OS project dies out, leaving the software unmaintained. However, this problem is not specific to the open source community. A commercial product can also die out if the company behind it goes bankrupt, or decides that it is no longer profitable to keep on maintaining the product.

Most of the mainstream OS frameworks used in the project worked well in agile development. Especially Spring and Hibernate supported the need for constant refactoring by being easy to configure and change. Spring Framework, which is a Java EE framework for enterprise applications, supports the test-driven approach by making all parts of the application automatically available for unit tests. Spring has been designed to support writing easily testable code, and all the components could easily be retrieved for testing by using the application context provided by Spring. The Spring application context was also convenient when setting up and refactoring the application architecture, since inter-object relationships could be set up from a single configuration file instead of hard-coding the object instantiations all around the software code.

Perhaps even more notably than Spring, Hibernate excelled when refactoring code. Hibernate is an object/relational mapping framework that allows developers to map their objects directly into a database, with no need to write database queries by hand. Because the Hibernate Annotations were used, the database mappings could be changed at the same time when the data object code itself was changed. Because there was no need to update separate configuration files, the database mapping configuration stayed up-to-date at virtually all times. Indeed, separate configuration files were identified as a possible problem source when refactoring. With a single framework, the maintaining

work should not be too large a problem. However, with several frameworks integrated to work together, multiple configuration files might need to be maintained when a change is made. This can cause problems because some of the required updates can easily be forgotten. In addition, because configuration related to a single functionality can be separated to many configuration files, the source of a possible problem can be hard to locate.

Although software frameworks contain features that support agile development practices, most of the agile practice support comes from the software tools used in the development process itself. The Eclipse IDE is a perfect example of an open-source development environment that is an invaluable aid in agile software development. Starting from the code completion, code browsing and automatic import management features, coding in Eclipse is much more efficient than coding with a normal text editor. Furthermore, for agile development, Eclipse contains features such as test case skeleton generation, integrated test suite running and automatic code refactoring tools. With the refactoring tools it is possible to rename a method or a class and automatically update all the invocations. With these tools, the effort required for applying large structural changes is diminished.

Even though most of the selected open-source solutions worked well in the project, and several of them supported the agile development practices well, there are still problems with the adoption of new software solutions. Foremost of all, the learning curve required to start using an application framework is still high, as well with open-source solutions as with commercial products. In this area, open-source solutions do have the drawback that their documentation can be inadequate. Some open source projects have exceedingly good documentation, but others might not have any documentation at all. Generally, with commercial products, at least some sort of documentation exists. Because of this, adopting an OS solution can possibly require a vast amount of exploratory work to find out how the software works and how it is used. Of the selected open-source software, XDoclet 2 contained virtually no documentation at all and the documentation of both Maven and Struts was partly incomplete and disorganised. The other solutions were either adequately or well documented. The findings here corresponding to documentation generally adhere with the basic principle for selecting open-source software – the larger and more widely used the product, the better the documentation. For Struts, which is a widely used product, there exists many other sources of documentation, such as books and articles published by other authors.

9.3 Agile Development Methods

The selected agile development practices performed with varying degrees. Most of the practices were adopted well, and the team managed to carry out the project using an

extremely lightweight process. However, the size of the team and the project must be taken into account when analysing the agile methods. Managing the activities of three persons can be easily coordinated regardless of the process; in contrast, managing a ten-man agile team requires stricter adherence to the selected processes and practices.

In the executed project, the selected subset of Extreme Programming practices was found to be effective in developing the system. The principles of simple design and refactoring allowed the team to quickly produce working drafts of the system for the customer to test. To mitigate the possible quality problems from simple initial design, the testing and refactoring practices have been used successfully to uphold the quality of the code. However, the need for some initial design has been identified for some of the core components in larger systems to avoid the need for massive structural changes later on.

Perhaps not surprisingly, the most problematic of the selected XP practices was the testing practice. The practice of writing all unit tests first does not come easily to developers, even though the justifications for the practice itself are well-based in theory. Developers feel that the actual implementation code is too easy to write at the same time when coding the skeletons required for creating the tests. However, the author has heard comments that once developers have used the practice successfully for a while, they would not like to switch away from it.

9.4 Summary

As a quick summary, mature open-source solutions and agile development methods can work well together, with a net effect of enhanced productivity and low development costs. However, no silver bullet is provided by either the open source community or the agile methodologies. There are still risks in using open-source software that is not widely used. Furthermore, the agile methodologies are still relatively young and not extensively studied. Fortunately the open source community is gaining foothold in the commercial world and studies are being made on the agile development practices. It is the author's belief that open-source software will be used more often in software projects that will be managed using agile processes.

Bibliography

A. Aarsten, D. Brugali, and G. Menga. Patterns for Three-Tier Client/Server Applications. *Proceedings of the Third Conference on the Pattern Languages of Programs (PLoP'96)*, Sep 1996.

P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. *Agile Software Development Methods - Review and Analysis*. VTT, 2002. ISBN 951-38-6009-4.

AgileAlliance. Agile Alliance. Internet, 2006. URL `http://www.agilealliance.org/`. Referenced 26.11.2006.

S.I. Ahamed, Alex Pezewski, and Al Pezewski. Towards framework selection criteria and suitability for an application framework. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, pages 424–428, 2004. doi: 10.1109/ITCC.2004.1286492.

E. Altendorf, M. Hohman, and R. Zabicki. Using J2EE on a large, Web-based project. IEEE *Software*, 19(2):81–89, Mar-Apr 2002. doi: 10.1109/52.991368.

AOPMetrics. AOP Metrics: Common metrics suite tool for Java and AspectJ. Internet, 2006. URL `http://aopmetrics.tigris.org/`. Referenced 24.11.2006.

Apache. Apache HTTP Server, 2006. URL `http://httpd.apache.org/`. Referenced 6.11.2006.

J. Arthur and S. Azadegan. Spring framework for rapid open source J2EE Web application development: a case study. IEEE *Proceedings of the Sixth International Conference on Sofware Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SPND/SAWN'05)*, pages 90–95, May 2005. doi: 10.1109/SNPD-SAWN.2005.74.

K. Auer and R. Miller. *Extreme Programming Applied: Playing to Win*. Pearson Education, Inc., 2002. ISBN 0-201-61640-8.

A. Balogh, G. Verró, D. Varró, and D. Pataricza. Compiling model transformations to EJB3-specific transformer plugins. *Proceedings of the 2006 ACM symposium on Applied computing SAC '06*, pages 1288–1295, Apr 2006.

C. Bauer and G. King. *Hibernate in Action*. Manning Publications Co., Aug 2004. ISBN 1932394-15-X.

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. ISBN 0201616416.

K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development. Internet, 2001. URL `http://www.agilemanifesto.org/`. Referenced 20.9.2006.

C.A. Berry, J. Carnell, B.J. Matjaz, M.M. Kunnumpurath, N. Nashi, and S. Romanosky. J2EE *Design Patterns Applied*. Wrox Press Ltd., 2002. ISBN 1-861005-28-8.

E. Bertin and P. Lesieur. Which architecture for integrated services? *International conference on Networking and Services, 2006. ICNS '06*, Jul 2006. doi: 10.1109/ICNS.2006.120.

W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns*. Wiley Computer Publishing, 1998. ISBN 0-471-19713-0.

D. Casal. Advanced software development for Web applications (TSW0505). Technical report, JISC Technology and Standards Watch, Dec 2005. URL `http://www.jisc.ac.uk/uploaded_documents/jisctsw_05_05pdf.pdf`. Referenced 19.9.2006.

S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. IEEE *Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

M.C. Chu-Carroll, D. Shields, and J. Wright. Version Control: A Case Study in the Challenges and Opportunities for Open Source Software Development. *Proceedings of the 2nd Workshop on Open Source Software Engineering*, May 2002.

A. Cockburn. *Agile Software Development*. Pearson Education, Inc., 2002. ISBN 0-201-69969-9.

Commons. Jakarta Commons. Internet, 2006. URL `http://jakarta.apache.org/commons/`. Referenced 14.11.2006.

CVS. CVS Wiki. Internet, 2006. URL `http://ximbiot.com/cvs/wiki/`. Referenced 14.11.2006.

J. Dedrick. An Exploratory Study into Open Source Platform Adaption. *Proceedings of the 37th Hawaii International Conference on System Sciences*, pages 5–8, Jan 2004. doi: 10.1109/HICSS.2004.1265633.

Eclipse. Eclipse - an open development platform, 2006. URL `http://www.eclipse.org/`. Referenced 2.11.2006.

D. Geer. Eclipse becomes the Dominant Java IDE. IEEE *Computer*, 38(7):16–18, Jul 2005. doi: 10.1109/MC.2005.228.

J. Goodwill. *Mastering Jakarta Struts*. Wiley Publishing, Inc., 2002. ISBN 0-471-21302-0.

Hibernate. Hibernate, 2006. URL `http://www.hibernate.org/`. Referenced 2.11.2006.

J. Highsmith and A. Cockburn. Agile software development: the business of innovation. *Computer*, 34(9):120–127, Sep 2001. doi: 10.1109/2.947100.

P. Hodgetts. Refactoring the Development Process: Experiences with the Incremental Adoption of Agile Practices. *Proceedings of the Agile Development Conference (ADC'04)*, pages 106–113, 2004. doi: 10.1109/ADEVC.2004.17.

M. Huo, J. Verner, L. Zhu, and M.A. Babar. Software Quality and Agile Methods. *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 520–525, 2004. doi: 10.1109/CMPSAC.2004.1342889.

A. Hussey and D. Carrington. Comparing the MVC and PAC architectures: a formal perspective. *IEE Proceedings - Software Engineering*, 144(4):224–236, Aug 1997.

JavaEE. Java EE at a Glance. Internet, 2006. URL `http://java.sun.com/javaee/`. Referenced 15.11.2006.

R. Johnson. *Expert One-on-One:* J2EE *Design and Development*. Wiley Publishing, Inc., 2003. ISBN 0-7645-4385-7.

R. Johnson. J2EE development frameworks. IEEE *Computer*, 38(1):107–110, Jan 2005a.

R. Johnson. Introduction to the Spring Framework. Internet, May 2005b. URL `http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework`. Referenced 14.11.2006.

R.E. Johnson. Components, Frameworks, Patterns. *Proceedings of the 1997 symposium on Software reusability*, pages 10–17, Feb 1997a.

R.E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40 (10):39–42, Oct 1997b. ISSN 0001-0782.

C. Larman. *Applying* UML *and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, Inc., 2 edition, 2002. ISBN 0-13-092569-1.

J. Locke. *Open Source Solutions for Small Business Problems*. Charles River Media, Inc., 2004. ISBN 1-58450-320-3.

Q.H. Mahmoud. Servlets and JSP Pages Best Practices. Sun Developer Network (SDN), Mar 2003. URL `http://java.sun.com/developer/technicalArticles/javaserverpages/servlets_jsp/index.html`. Referenced 6.11.2006.

V. Mahnic and S. Drnovscek. Agile Software Project Management with Scrum. *EUNIS 2005 Conference Programme*, Jun 2005. URL `http://www.mc.manchester.ac.uk/eunis2005/medialibrary/papers/paper_194.pdf`. Referenced 22.11.2006.

F. Maurer and S. Martel. Extreme programming: Rapid development for Web-based applications. IEEE *Internet Computing*, 6(1):86–90, Jan-Feb 2002. doi: 10.1109/4236.989006.

Maven. Apache Maven. Internet, 2006. URL `http://maven.apache.org/`. Referenced 14.11.2006.

P. McBreen. *Questioning Extreme Programming*. Pearson Education, Inc., 2003. ISBN 0-201-84457-5.

R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, Sep 2001. ISBN 0-596-00226-2.

Netcraft. Web Server Survey, Nov 2006. URL `http://news.netcraft.com/archives/2006/11/01/november_2006_web_server_survey.html`. Referenced 6.11.2006.

Sun Developer Network. Core J2EE Patterns. Internet, 2001-2002. URL `http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html`. Referenced 14.11.2006.

C.F. Ngolah and Y. Wang. Exploring Java Code Generation Based on Formal Specifications in RTPA. *Canadian Conference on Electrical and Computer Engineering, 2004*, 3: 1533–1536, May 2004.

OSI. Open Source Initiative, 2006. URL `http://www.opensource.org/`. Referenced 10.10.2006.

J. Parviainen. J2EE Presentation Layer Frameworks. Master's thesis, Helsinki University of Technology, Espoo, Finland, Feb 2006.

Y. Ping, K. Kontogiannis, and T.C. Lau. Transforming legacy Web applications to the MVC architecture. *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice, 2003*, pages 133–142, Sep 2003. doi: 10.1109/STEP.2003.35.

L. Rising and N.S. Janoff. The Scrum software development process for small teams. IEEE *Software*, 17(4):26–32, Jul-Aug 2000. doi: 10.1109/52.854065.

M. Ruffin and C. Ebert. Using open source software in product development: a primer. IEEE *software*, 21(1):82–86, Jan-Feb 2004. ISSN 0740-7459. doi: 10.1109/MS.2004.1259227.

S.R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 5th edition, 2002. ISBN 0-07-112263-X.

K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2003. ISBN 0-7356-1993-X.

Scrum. Control Chaos: Scrum. Internet, 2006. URL `http://www.controlchaos.com/`. Referenced 22.11.2006.

D.M. Selfa, M. Carrillo, and M. Del Rocio Boone. A Database and Web Application Based on MVC Architecture. *16th International Conference on Electronics, Communications and Computers (CONIELECOMP 2006)*, Feb 2006. doi: 10.1109/CONIELECOMP.2006.6.

N. Serrano and I. Ciordia. Ant: automating the process of building applications. IEEE *Software*, 21(6):89–91, Nov-Dec 2004. doi: 10.1109/MS.2004.33.

J.F. Smart. An introduction to Maven 2 - How applied best practices can optimize the Java build process. Internet, May 2005. URL `http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html?lsrc=maven-users`. Referenced 14.11.2006.

J.W. Spence. There Has to Be a Better Way! *Agile Conference, 2005. Proceedings*, pages 272–278, Jul 2005. doi: 10.1109/ADC.2005.47.

Spring. Spring Framework. Internet, 2006. URL `http://www.springframework.org/`. Referenced 14.11.2006.

Struts. Apache Struts, 2006. URL `http://struts.apache.org/`. Referenced 2.11.2006.

R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE *Transactions on Software Engineering*, 29(4):297–310, Apr 2003. doi: 10.1109/TSE.2003.1191795.

Subversion. Subversion. Internet, 2006. URL http://subversion.tigris.org/. Referenced 14.11.2006.

E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing Server-Side Distribution. *Proceedings of the 18th* IEEE *International Conference on Automated Software Engineering (ASE'03)*, pages 130–141, Oct 2003. doi: 10.1109/ASE.2003.1240301.

E. VanDoren. Cyclomatic Complexity. Internet, Jan 1997. URL http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html. Referenced 27.11.2006.

M.W. Whalen and M.P.E Heimdahl. On the Requirements of High-Integrity Code Generation. *Proceedings of the 4th* IEEE *International Symposium on High-Assurance Systems Engineering, 1999*, pages 217–224, Nov 1999. doi: 10.1109/HASE.1999.809497.

L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the Case for Pair Programming. IEEE *Software*, 17(4):19–25, Jul-Aug 2000. doi: 10.1109/52.854064.

L. Williams, W. Krebs, and L. Layman. Extreme Programming Evaluation Framework for Object-Oriented Languages – Version 1.2. *North Carolina State University, Raleigh, NC Computer Science TR-2004-1, January*, 5, 2004a.

L. Williams, W. Krebs, L. Layman, and A. Antón. Toward a Framework for Evaluating Extreme Programming. *Proceedings of the Eighth International Conference on Empirical Assessment in Software Engineering (EASE 04)*, 2004b.

XDoclet1. XDoclet: Attribute-Oriented Programming, 2005. URL http://xdoclet.sourceforge.net/xdoclet/index.html. Referenced 2.11.2006.

XDoclet2. XDoclet2, 2006. URL http://xdoclet.codehaus.org/. Referenced 2.11.2006.

L. Zhang, S. Akifuji, K. Kawai, and T. Morioka. Comparison Between Test Driven Development and Waterfall Development in a Small-Scale Project. *Proceedings of the 7th International Conference on XP (LNCS 4044)*, pages 211–212, Jun 2006.

Appendix A

Shodan Input Metric Survey

The questions presented in the subjective XP adherence metric survey are shown in table A.1. The questions have been taken from [Williams et al., 2004a]. For each question, the respondents were asked to use the following scale:

10	Fanatic	100%	4	Common	40%
9	Always	90%	3	Sometimes	30%
8	Regular	80%	2	Rarely	20%
7	Often	70%	1	Hardly ever	10%
6	Usually	60%	0	Disagree with using this practice	0%
5	Half 'n Half	50%			

TABLE A.1: Survey Questions

Practice	**Description**
Automated Unit Tests	You run automated unit test (such as JUnit) each time you make a change. What % of your changes are tested with automated unit tests before they are checked in?
Test First Design	Write test cases, then the code. The test case is the spec. What % of your code line items were written AFTER an automated test was developed for the corresponding scenario?
Refactoring	Rewrite code that 'smells bad' to improve future maintenance and flexibility without changing its behaviour. What % of the time do you stop to cleanup code that has already been implemented without changing functionality?

Practice	**Description**
Short Releases	You have frequent smaller releases instead of larger, less frequent ones. This lets the customer see how it's going and lets you get feedback. How close are you to having releases that are about 3 months with interim iterations of a couple weeks?
Coding Standards	Do you have and adhere to team coding standards? Besides brace placement, this may include things like logging and performance idioms. How often do you follow your team standards?
Collective Ownership	You can change anyone's code and they can change yours. You don't get stuck when the expert is busy on vacation. People know many parts of the system. How often do people change code they did not originally write?
Simple Design	Keep it simple at first; do the simplest thing that could possibly work. You don't follow the philosophy of "I'll include this because the customer might possible need it later" even though the feature isn't in the requirements. Also, you do not spend a lot of time on design documents. How often do you succeed in 'Keeping it Simple'?

Appendix B

Objective XP Adherence Metrics

The measurements for the objective XP adherence metric are shown in table B.1. The measurements have been taken from [Williams et al., 2004a].

TABLE B.1: Objective XP Adherence Metrics

Metric	**Practice**	**Description**
Unit Test Runs / Person Day	Testing	While a set of test cases may be available it is the developer's responsibility to run them. This metric determines how often test suites are run.
Test LOC / Source LOC	Testing	Examine the ratio of test lines of code to source lines of code as a relative measure of how much test code is written by the team.
Release Length	Short Release	Measure the release length in order to gauge the extent to which the project adheres to the 'short release' XP practice. While there is no defined length for what qualifies a short release, XP advocates a maximum of 3 month release period.
Iteration Length	Short Release	Assess if the team is using short iterations. XP employs short iterations to receive continuous feedback on product development from all stakeholders. XP advocates an iteration length of at most three weeks.

Appendix C

Comparison Metrics

This appendix lists the metric values used for comparison purposes in section 8.1.2. The values have been calculated for four systems that are described in table C.1. The values themselves are shown in table C.2. The original values were presented in differing formats, which is reflected in the table layout. In addition, not all of the metrics were calculated for systems C and D.

TABLE C.1: Compared Systems

System	Language	Classes	Source
A	C++	634	[Chidamber and Kemerer, 1994]
B	Smalltalk	1459	[Chidamber and Kemerer, 1994]
C	C++	405	[Subramanyam and Krishnan, 2003]
D	Java	301	[Subramanyam and Krishnan, 2003]

TABLE C.2: Comparison Metrics

Metric	System	Mean	Median	Min	Max
WMC	A	-	5	0	106
	B	-	10	0	346
	C	7.07	-	0	219
	D	12.15	-	0	132

Metric	System	Mean	Median	Min	Max
DIT	A	-	1	0	8
	B	-	3	0	10
	C	2.36	-	0	5
	D	1.02	-	0	5
NOC	A	-	0	0	42
	B	-	0	0	50
CBO	A	-	0	0	84
	B	-	9	0	234
	C	2.81	-	0	25
	D	2.94	-	0	29
RFC	A	-	6	0	120
	B	-	29	3	422
LCOM	A	-	0	0	200
	B	-	2	0	17