ESPOO 2007 VTT RESEARCH NOTES 2387

Katja Henttonen

Stylebase for Eclipse

An open source tool to support the modeling of quality-driven software architecture

VTT TIEDOTTEITA – RESEARCH NOTES 2387

Stylebase for Eclipse

An open source tool to support the modeling of quality-driven software architecture

Katja Henttonen

ISBN 978-951-38-6925-0 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O.Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax +358 20 722 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Technical editing Leena Ukskoski

Text preparing Kirsi-Maarit Korpi

Cover picture http://www.flickr.com/photos/ilovetrance/298742832/

Henttonen, Katja. Stylebase for Eclipse. An open source tool to support the modeling of quality-driven software architecture [Tyylikanta Eclipseen. Avoimen lähdekoodin työkalu tukemaan laatuohjatun ohjelmistoarkkitehtuurin mallintamista]. Espoo 2007. VTT Tiedotteita – Research Notes 2387. 61 p. + app. 15 p.

Keywords Eclipse, open source software, modeling, quality-driven software, software architecture, basecode, database design, stylebase, open source community

Abstract

Open source software has gained a lot of well-deserved attention during the last few years. Eclipse is one of the most successful open source communities providing an open development environment and an application lifecycle platform. Eclipse is a vendor-neutral platform for integrating tools and services. My thesis work is a case study on contributing to Eclipse. The contribution is a software architecture tool called "Stylebase for Eclipse" which is implemented as an extension a.k.a. *plug-in* to Eclipse.

Quality-driven architecture design is an approach to software architecture design which emphasizes the importance of qualities. Qualities are non-functional characteristics of a software system such as security or maintainability. Stylebase is a knowledge base of *software patterns* and *architectural styles*. It stores information that helps a software architect in selecting patterns that best support the desired quality goals. Stylebase for Eclipse is a tool for browsing and maintaining the stylebase. The purpose of the tool is to improve the quality of design and increase information sharing and re-use of architectural models in development teams.

In the case study, the plug-in is first developed and, after that, a new open source community is formed around the plug-in project. In order to comply with the open source development model, *modularity* is treated as the most important non-functional requirement. In community building phase, efforts are concentrated on marketing the new open source project and creating a good technical infrastructure for it.

The most interesting experiences gained during the study are related to various aspects of open source development. They are – among others – re-using code from other projects, licensing issues, tools to facilitate distributed development , and attracting new users and developers.

Henttonen, Katja. Stylebase for Eclipse. An open source tool to support the modeling of quality-driven software architecture [Tyylikanta Eclipseen. Avoimen lähdekoodin työkalu tukemaan laatuohjatun ohjelmistoarkkitehtuurin mallintamista]. Espoo 2007. VTT Tiedotteita – Research Notes 2387. 61 s. + liitt. 15 s.

Keywords Eclipse, open source software, modeling, quality-driven software, software architecture, basecode, database design, stylebase, open source

Tiivistelmä

Avoimen lähdekoodin ohjelmistot ovat saanet paljon ansaittua huomiota viime vuosina. Eclipse on yksi menestyneimmistä avoimen lähdekoodin yhteisöistä, joka tarjoaa avoimen kehitysympäristön ja sovelluskehyksen. Eclipse on toimittajariippumaton integrointialusta työkaluilulle ja palveluille. Opinnäytetyöni on tapaustutkimus, jossa tehdään *kontribuutio* eli lahjoitetaan kehitystyötä Eclipse-yhteisölle. Kontribuutio on ”*Stylebase for Eclipse*” -niminen työkalu ohjelmistoarkkitehdeille, joka toteutetaan Eclipse-laajennoksena.

Laatuohjattu arkkitehtuurisuunnittelu on lähestymistapa, joka painottaa laatuominaisuuksien merkitystä ohjelmistoarkkitehtuurin suunnittelussa. Laatuominaisuuksilla tarkoitetaan ohjelmiston ei-toiminnallisia piirteitä, esimerkiksi tietoturvallisuutta tai ylläpidettävyyttä. Stylebase eli *tyylikanta* on tietovarasto, joka sisältää arkkitehtuurityylejä ja suunnittelumalleja. Tyylikantaan tallennetun tiedon avulla ohjelmistoarkkitehti osaa valita ne tyylit ja mallit, jotka parhaimmin tukevat määriteltyjä laatutavoitteita. Stylebase for Eclipse on työkalu tyylikannan selaamiseen ja hallintaan. Työkalun tarkoitus on parantaa suunnittelutyön laatua sekä edistää tiedon vaihtoa ja arkkitehtuurimallien uudelleenkäyttöä kehitystiimeissä.

Tutkimuksessa rakennetaan ensin laajennos ja sitten perustetaan oma avoimen lähdekoodin yhteisö jatkamaan laajennoksen kehitystyötä. *Modulaarinen* rakenne on laajennoksen tärkein ei-toiminnallinen ominaisuus, koska se luo edellytykset avoimen lähdekoodin kehitystyölle. Yhteisön perustamisvaiheessa keskitytään projektin markkinointiin sekä sen tarvitseman infrastruktuurin rakentamiseen.

Mielenkiintoisimmat kokemukset ja tulokset liittyvät avoimen lähdekoodin kehityksen eri piirteisiin. Näitä ovat mm. ohjelmakoodin uudelleenkäyttö, lisensiointikysymykset, hajautetun kehitystyön apuvälineet sekä uusien käyttäjien ja kehittäjien löytäminen.

Abbreviations

ANSI	American National Standards Institute
COSI	Co-development of Inner and Open Source in Software Intensive Products (a research project)
CPL	Common Public License
FLOSS	Free/Libre/Open Source Software
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DTD	Document Type Definition
eCore	The meta model used by EMF
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
GUI	Graphical User Interface
GNU	Gnu is Not Unix (a recursive acronym)
GPL	GNU General Public License
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
ITEA	The International Technology Education Association
IEEE	Institute of Electrical and Electronics Engineers
JBoss	A Java-based, open source application server
JDBC	Java Database Connectivity
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MVC	Model-View-Controller (an architecture pattern)
MQ	Message Queue
MyISAM	A storage engine for MySQL database
MySQL	Database management system owned by MySQL Ab.
OMG	Object Management Group
OSS	Open Source Software
OSI	Open Source Initiative

PC	Personal Computer
QADA	Quality-Driven Architecture Design and Analysis
SQL	Structured Query Language
SQL-92	Third revision of SQL standard
SWT	Standard Widget Toolkit
XMI	XML Metadata Interchange
XML	Extensible Markup Language
UML	Unified Modeling Language

Contents

1. Introduction

Open source software (OSS) has seen its popularity grow among software developers, communities, and media for over a decade. Today, successful open source products include Apache and MySQL and are dominating the market and big software companies (such as IBM and Sun Microsystems) have made significant investments in open source development projects. Research community has also been active in studying the subject as demonstrated, for example, by the recent series of OSS conferences (Heizman 2003, Deitel 2007). The open source movement is hard to understand unless one sees the two sides of it. On the one hand, it is a largely technically oriented phenomenon, and on the other hand, it is something that challenges conventional organizational and business viewpoints (Helander & Mäntymäki 2006, 1).

Among the most successful open source projects, Eclipse is an open development platform and application framework for building software. This Bachelors thesis is a case study on contributing to Eclipse. The contribution to the Eclipse community is a tool for managing a software architectural knowledge base, i.e. stylebase. The tool is implemented as an Eclipse extension and published under a well-known open source license. The result is available at http://stylebase.sourceforge.net. In this report, contributing to Eclipse is studied both from technical and organizational perspectives. First comes the technical task of developing the extension and then comes the organizational task of founding a new open source community around it.

The tool itself is based on the original idea of an architectural knowledge base which was published in IEEE Transactions journal (Niemelä et al.) in 2003. The first tool for maintaining such a knowledge base – was implemented as an extension to a closed-source modeling tool (Merilinna 2005). This work brings the concept into the open source world for the first time.

In this section, I first provide definitions to the main terminology which is essential to understanding the study. The motivation of the research is presented in order to highlight both the need for an architectural knowledge management tool and the benefits of going to open source. Thereafter, the objectives of the research are presented together with respective metrics. Research problems, the scope of the study, and the selected approach are discussed. The section ends with an overview on the structure of this document.

1.1 Main terminology

The *model* is described as an abstract presentation of a software system (Selic 2003a). In this report, *pattern's data model* refers to a presentation which describes pattern's components, their primary attributes, and relationships. Such presentation is typically, but not necessarily, an UML diagram saved in XML format.

Model-Driven Development (MDD) is an approach to software development that focuses on models – not programs – as primary software artifacts. Modeling helps to analyze complex problems and their potential solutions before going through the effort and expense of building the final product. (Selic 2003b.)

The *software architecture* is a structure (or structures) of a program or computing system. The structures consist of software components plus their properties and interrelationships. Software architecture controls program's evolution. (Bass et al. 2004, 3.)

The quality-driven architecture design is an approach for software architecture design which emphasis the significance of *quality attributes*. Quality attributes are gathered, categorized, and documented and then the acquired knowledge is used in architecture design. The approach is complemented by *quality-driven architecture analysis,* which is a way of analyzing software architecture from the quality point of view. (Matinlassi et al. 2002, 13; Matinlassi 2006, 14.)

Quality attributes describe externally or internally observable quality-characteristics of a software component or system. (Niemelä & Immonen 2007.)

S*tylebase* is a reuse repository of software architectural styles and patterns. Stylebase is an essential part of the quality-driven software architecture design and analysis methodology. The idea is to improve the quality of software products by assisting software architects in selecting models that best meet the desired quality goals. (Matinlassi et al. 2002, 31.)

An *architectural pattern* expresses a fundamental structural organization schema for software systems, which consists of predefined subsystems and specifies their responsibilities and relations. In comparison to design patterns, architectural patterns are larger in scale. (Bushmann et al. 1996, 13.)

A *design pattern* describes a schema of communicating objects and their relations, which solves a general design problem in a particular context. While design patterns are not language dependent, they are based on practical solutions which have been implemented in mainstream programming languages. (Gamma et al. 1994, 2–4.)

Modularity is extent to which software is composed out of separate modules with minimal interactions between the modules. Modules are parts of software which operate independently, yet function "as a whole". Modules communicate with each other via predefined interfaces. Building a system to be modular produces many benefits, for example, modular software is easier to maintain. (Gershenson et al. 2003.)

Open source software is software which is available with its source code and under an open source license. Such a license permits anyone to study, change, improve, and redistribute the modified or unmodified version of the software (Open Source Initiative 2006). In this report, *free software* is used synonymously with open source software.

Community is a group of people with a shared interest or goal who get to know each other better over time. *An open source community* is a group of users and developers who are all interested in the development of a particular open source product and communicate with each other over Internet. (Goldman & Gabriel 2005, 32.)

Eclipse is an open source development platform and application framework for building software. Eclipse assist in creating, integrating, and utilizing software tools. Eclipse is also a community of users and developers who work together to improve the product. (Eclipse Foundation 2006a.)

Platform is defined as subsystems or technologies that provide a coherent set of functionality through interfaces and specified usage patterns. Subsystems that depend on the platform need not be concerned on details of how the functionality provided by the platform is implemented. (Miller & Mukerji 2003, 58.)

Plug-in is a bunch of code that enhances the functionality of an existing software program (IBM Corporation and others 2003), in this study, Eclipse.

1.2 Motivation

Collecting styles and patterns into a knowledge base encourages designers to use them, resulting in better utilization of existing software architectural know-how. The idea of stylebase relates essentially to quality-driven architecture design. The stylebase assists a software architect in selecting appropriate styles based on the quality requirements set for the system at hand. It can also be utilized when analyzing software architectures from the perspective of quality. Thus, the stylebase helps to improve the quality of software products and increase knowledge sharing and re-use in development teams. (Niemelä et al. 2003.)

A tool is needed for browsing and maintaining the stylebase. The first version of such a tool was implemented as an extension to a proprietary modeling environment called Tau/Developer in 2004 (Merilinna 2005). The decision limited potential users to those already using this particular, fairly expensive modeling tool. In order to gain a wider user base, an open source version of the tool is needed. The open source approach helps to disseminate the idea of stylebase, making VTT's research results available to much more people. By going to open source, external developers can be attracted to the project and code from other projects can be re-used.

The study shall accumulate knowledge in founding an open source community and keeping it active. The work forms a case study for an ITEA research project called COSI. COSI **C**o-development using inner & **o**pen source in **s**oftware **i**ntensive products) project aims to create strong awareness of the industrial usage of distributed collaborative software and open source. *Inner source,* which is not within the framework of my thesis work, refers to the application of open source development methods to closed source activities such as software development inside a company. My thesis work is part of a work package that aims to define the best practices for requirement analysis and architecture design management for open source software. A task lead by VTT focuses on the various dimensions of integration and quality assessment raised by open source development.

1.3 Problems, approach and limitations

The primary research problem address in my thesis can be stated as follows:

How to contribute to Eclipse by initiating a new plug-in project?

There are several ways of contributing to open source projects, e.g. fixing bugs and providing new features. In this case study, contributing to Eclipse is considered a twofold issue (Henttonen & Matinlassi 2007). On the one hand, a contribution, in this case a plug-in, needs to be developed, and on the other hand, it needs to get published to the community: users and developers (Beck & Gamma 2004). A good plug-in is modular, because the only way an open source project can mature is by allowing a number of people to participate in the development (Fleury & Lindfors 2001). The primary research question can thus be divided into the following sub questions.

How to design a modular Eclipse plug-in?

How to launch a successful open source community?

The study focuses on contributing to Eclipse by founding a new plug-in project. Other ways of contributing to Eclipse, e.g. committer contributions to the global Eclipse release, are not within the scope of this study.

According to literature, successful open source projects are generally not launched before their code already makes up a working application (Fogel 2005, 44; Weinstock & Hissam 2005, 156). Goldman and Gabriel (2005, 256) write:

> Remember that open source only works to incrementally improve what's there – if you only have design ideas that's what the community will improve. Someone needs to write a code for a minimal but working version before the community will contribute any code.

It was therefore decided that a code artifact is implemented *before* announcing the project. Mandatory requirements, which need to be implemented before announcing the project, are identified in the requirement specification phase.

Marketing activities are an essential part of the study. While developing modular plug-in is relatively straightforward (Gamma & Beck 2004), the acute question remains how to get the modular plug-in to the open source "market". In order to succeed, an open source project needs marketing just like any other product (Goldman & Gabriel 2005).

The research is essentially empirical in nature. Figure 1 illustrates the empirical part of the research process as a flow chart diagram.

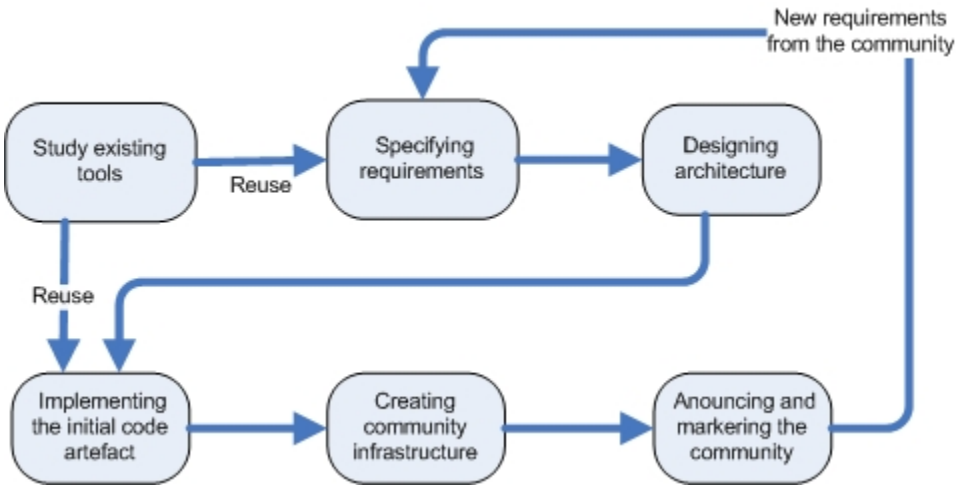

Figure 1. Research process (empirical part).

Literature is reviewed in order to make positive decisions during the study. Experiences are written down during the process and analyzed at the end of the study. The experience report shall discuss how the approaches recommended in the literature did work in practice. Further development is not in the scope of my thesis work. However, the thesis work includes collecting the feedback from the user community which shall then guide the further development of the Stylebase for Eclipse.

1.4 Research objectives and metrics

The concrete objectives of the research can be divided into technical, business-oriented, and organizational. The three primary research objectives and their respective metrics (Matinlassi & Henttonen 2007) are summarized in Tables 1–3.

Table 1. Technical Goal Summarized.

Objective	Develop an Eclipse compatible version of the stylebase maintenance tool
Expected Result	A modular Eclipse plug-in in Java
Metrics	Tool demonstration, criteria defined in requirement specification
Data Collection	Research log filled in during the development phase

Essential design decisions will be collected from the research log and reported and analyzed in this document. Unified Modeling Language (UML) is used to illustrate structure and behavior of the plug-in.

Table 2. Business Goal Summarized.

Objective	To initiate a plug-in project in the Eclipse community
Expected Result	Stylebase plug-in listed at Eclipse Plug-in Central
Metrics	The amount of downloads, the amount and quality of feedback and contacts from users
Data Collection	Project statistics, support and bug trackers, mailing lists, websites, emails received by the project administrator

Because users and developers are geographically far apart, an open source project needs good technical infrastructure for maintaining feedback loops (Goldman & Gabriel 2005, 175). Experiences on building such infrastructures are reported and analyzed in this document. Marketing efforts are described and compared to changes in the number of downloads and contacts.

Table 3. Organizational Goal Summarized

Objective	Deployment of active users in near future
Expected Result	Contributions to the development of the plug-in
Metrics	The amount of active users in the project, the quality and size of received contributions
Data Collection	Project statistics, source code repository, authors log

The third objective is not likely to be fully achieved before the thesis work is complete. Experiences on working with external contributors will be reported if available. Goldman and Gabriel (2005, 49) remind us that even successful open source projects grow relatively slowly and it is not realistic to expect a surge of volunteer developers joining the project right away.

1.5 Document structure

Section 2 provides the theoretical framework for the case study. It first discusses the importance of software architecture and then introduces design approaches which are essential to the motivations of the study. After that, an introduction is provided to open source software development, as well as Eclipse.

Section 3 describes the technical tasks related to the development of the Stylebase for Eclipse. Firstly, existing Eclipse extensions for pattern management were studied and then the work proceeded from requirement specification to database design and architecture development.

Section 4 describes primary tasks related to founding a new open source project. Firstly, one needed to choose and apply an open source license and build infrastructure required for open source development. Once the project had been made public, marketing activities ware carried out for attracting users and developers. The chapter ends with a subsection which takes a look at feedback and code contributions received from the community.

Section 5 discusses acquired experiences, near-future research plans, and certain aspects of the thesis process. The section also contains discussion on what was learned from the experiences and how well the objectives of the study were achieved.

Conclusions in section 6 close the thesis. The last section summarizes the results of the research, providing answers to the research questions.

The thesis is complemented by the following appendices. Appendix 1 contains a comparison table of existing pattern management tools which were studied as part of the thesis work. Appendix 2 provides an elaborate requirement specification of the Stylebase for Eclipse tool. Appendix 3 contains detailed description of underlying database schema. Appendix 4 presents the concrete architecture of the tool from structural point of view by providing class diagrams and composite structure diagrams of each component.

2. Architecture design, open source and Eclipse

This section introduces key concepts that are essential in understanding the title of the thesis work, as well as the study itself. The first subsection views the meaning of *software architecture* and provides an introduction to *modeling*. After that, it discusses a *quality-driven* approach to architecture development and explains the role of *stylebase* in supporting quality-driven design and analysis. The most important aspects of *open source* software development are summarized in the second subsection. The third subsection takes a look at *Eclipse*, both as an open source product and a community.

2.1 Software architecture design

In the last several years, there has been growing use of the word "architecture" in the context of software development. According to Bass et al. (2004, 3) “software architecture of a program or computing system is the structure of structures of the system, which comprise software components, the externally visible properties of the software components and relations between them”. In the same way as construction architecture guides building of a house, the software architecture guides the implementation of a software system. Software architecture has significant impact of the properties of the software system as it influences the whole life cycle of the software and guides its evolution (Matinlassi 2006, 13–14).

According to Bass, Clements and Kazman (2004, 26–29) software architecture is important for the following three main reasons. (1) Software architecture represents a common abstraction of a system which different stakeholders can use as a basis of *communication*. (2) Software architecture manifests the *early design decisions* which have significant, long-term impact on the development and maintenance of the software system. This is the earliest point when these decisions can be analyzed. (3) Software architecture constitutes *transferable abstraction of a system* – a model which can be applied to other systems which exhibit similar requirements.

Software architecture design plays an important role in developing prime quality software products. The following sections introduce two interrelated approaches to software architecture design: architecture modeling and quality-driven architecture design.

2.1.1 Software architecture modeling

A model is a simplified image of a system. Models and modeling are an essential part of traditional engineering. Nobody would consider building a car or a sky scraper without first constructing several specialized models. Models help to understand a complex problem and its potential solutions through abstraction. Software systems, which are often among the most complex engineering systems, can also benefit from the use of modeling techniques. This requires that models are of high-quality, i.e. abstract, understandable, accurate, predictive, and inexpensive. (Selic 2003b.)

Software *architecture model* is a description of software structures, presented by one or many *architectural views*. A view is composed of one or more architectural diagrams and represents the whole software system from a particular perspective. For example, a view can be targeted to a specific stakeholder (e.g. customer, designer) or present only certain properties of a system (e.g. performance model). Views can abstract away details, and a single view can have several abstraction levels. Diagrams are developed in accordance with the conventions established by an associated *architectural view point*. There are no fixed set of views, view points, or abstractions levels, but all are defined by the architecture design method at hand. (IEEE Standards Committee 2000.)

Model-Driven Development (MDD) is an approach to software engineering which treats software models – not computer programs – as the focus and primary products of software. The major advantage of MDD is that, compared to source code, models are easier to specify, understand, and maintain. In some cases, modeling makes it possible for domain experts to produce systems without knowledge on underlying implementation technology. As models are less bound to the chosen computing technology, they are also less sensitive to evolutionary changes in that technology. MDD methods rely on automation and the benefits it brings. Models are not very useful if their only end-up in documentation as such documentation becomes easily out-dated. A key premise behind MDD is that concrete programs are automatically generated from corresponding models. (Selic 2003a.)

Object Management Group (OMG) is driving *Model-Driven Architecture* (MDA) initiative. MDA is a formalization of the MDD approach. As defined by the OMG (2004), MDA is "a way to organize and manage enterprise architectures, supported by automated tools and services, for both defining the models and facilitating transformations between different model types". MDA Manifesto (Booch et al. 2004) introduces the three fundamental ideas of MDA (Figure 2) as follows. (1) The focus of software development should shift from the technology domain to the problem domain. Models should represent problems rather than be used as graphic syntax of programming languages. (2) Automated tools must be used to transform domain-

specific models into implementation code. (3) Standards are efficient boosters of technological progress. Open source development ensures the consistency of standards and encourages vendors to use them.

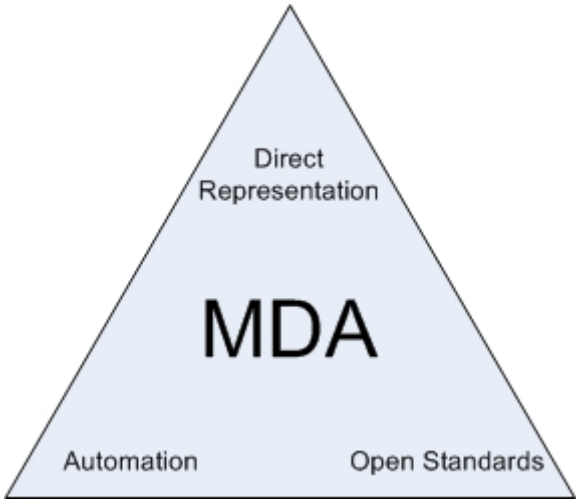

Figure 2. The tenets of MDA (Booch et al. 2004).

In MDA, system functions are defined in a platform independent fashion. On the top-level there is a platform independent model (PIM) which can be automatically transformed to a platform specific model (PSM) and finally into running code (Miller & Mukerji 2003). In this context, the term "platform" can be understood as defined by Frankel (2003). According to him, platform is a formatting technology (e.g. XML DTD), 3rd of 4th generation programming language (e.g. Java), distributed component middle-ware (e.g. CORBA) or messaging middle-ware (e.g. WebSpare MQ).

Unified Modeling Language (UML) serves as one of the corner stones of MDA. UML is a widely-used graphical modeling language that uses standardized diagrams to express the structure and behavior of software systems. The first version of the UML did not have the rigor and precision required to support MDA to its full extent. New modeling features were added to UML 2.0 in order to better meet with the requirements of MDA. (Selic 2003a.)

Adoption of model-driven methods in software engineering has faced resistance to change, mostly because code-centric approach is deeply rooted in the minds of software specialists (Matinlassi 2006, 43). However, according to Selic (2003b), MDD/MDA can bring significant reliability and productivity benefits that software needs to become a stable engineering discipline. This is assuming that the methods are applied correctly and quality issues are taken into account.

2.1.2 Quality-Driven Architecture Design

Quality issues play a significant role in the development of software products. Therefore, quality of software should be evaluated in the earliest possible phase, i.e. from the descriptions of the software architecture. Quality-driven architecture design emphasizes the importance of addressing quality requirements in architecture design phase. In order to evaluate quality on the architectural level, quality properties have to be defined and represented in architectural models.

Quality attributes are non-functional quality characteristics of a software system. According to Niemelä and Immonen (2007), they can be divided into two main categories. (1) Execution qualities, such as security and usability, are observable at run time. (2) Evolution qualities, such as extensibility and scalability, embody in the static structure of the software system.

Quality-Driven architecture design relies on the assumption that architectural styles and patterns, and also design patterns, embody different quality attributes. When patterns are applied in the architecture, quality-characteristics of the selected patterns are reflected to the entire software architecture (Matinlassi et al. 2002, 59–60). Architectural styles and patterns are re-usable designs applied in many contexts. They offer standard, tested solutions to common problems. Design patterns address coding problems, for example, how to convert an interface class into another one when interfaces are incompatible (Gamma et al. 1994, 22). Architecture patterns are larger in scale and address challenges in architecture design, for example, how to separate user interface from application logic (Bushmann et al. 1996, 13–14, 125).

As convertional modeling approaches do not support representation of quality requirements, VTT has developed a quality-driven architecture design and analysis methodology (QADA®[1]). QADA® provides a systematic way to transform quality requirements into software architecture. Styles and patterns are used as a guide to carry out quality requirements in architectural description with a documented design rationale. (Matinlassi et al. 2002, 13)

QADA® contributes to quality-driven architecture modeling – among many other things – by providing a predefined set of views which each use a set of diagrams for representing software architecture from a particular view point. In the QADA® method, there are four viewpoints: structural, behavioral, deployment, and development. The *structural view* records the hierarchical structure of architectural elements: components, their relationships, and responsibilities. The *behavior view* is used to describe actions that are produced, ordered, and synchronized by the system. The *deployment view* clusters conceptual components into deployment units and describes allocation of these

[1] QADA® is a registered trademark of VTT Technical Research Centre of Finland.

units into physical computing nodes. The *development view* also includes a business model as it defines who uses and who is responsible for services provided. Each viewpoint has two levels of abstraction: conceptual and concrete. In the *conceptual architecture design* phase, structure, behavior, and deployment of the system is modeled and documented on abstract level. In *the concrete architecture design* phase, concrete software components are defined in detail using the architecture descriptions produced in conceptual design as input. (Matinlassi et al. 2002, 25–29; Matinlassi & Kalaoja 2002.)

QADA also offers several methods for evaluating quality properties of a software system. The main focus is on evolution qualities. Evaluation is based on scenario-based analysis methods. This means that (1) evaluation scenarios are defined when quality requirements are defined, (2) they are taken into account while architecting and (3) evaluation is based on architectural models.

Figure 3 illustrates different dimensions of the QADA® methodology. Many areas of the method are not covered in this report.

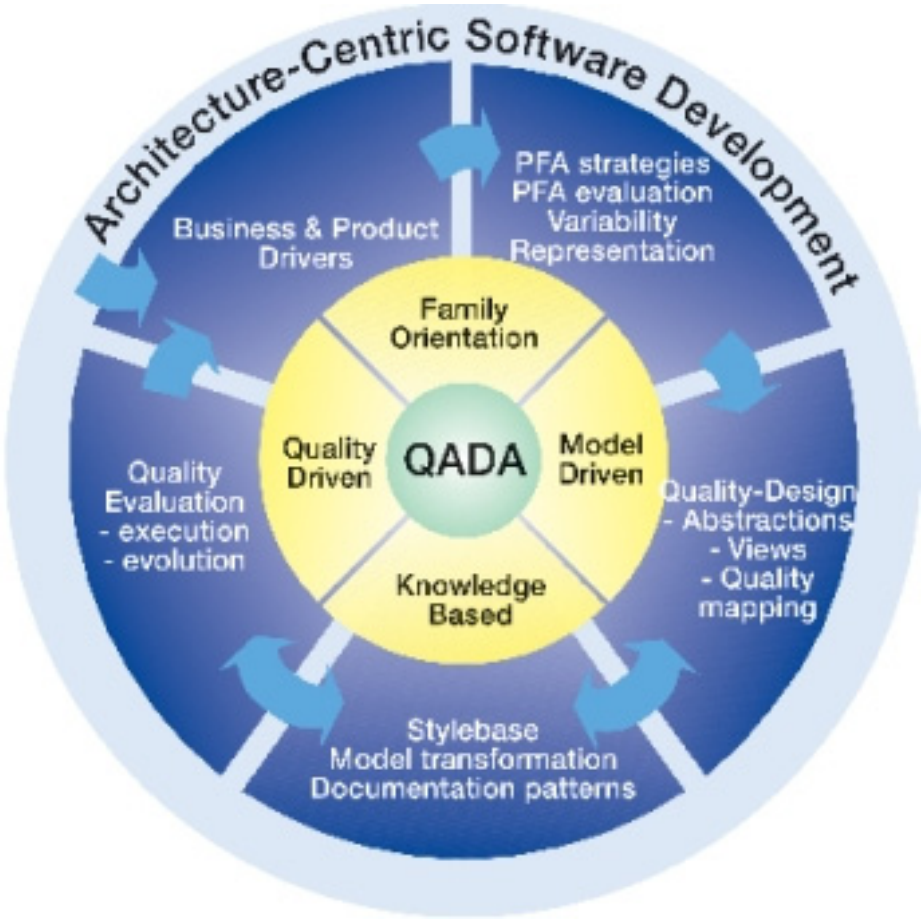

Figure 3. QADA® methodology (Niemelä et al. 2004).

As seen in the above figure, the stylebase (i.e. a knowledgebase of software architectural styles and patterns) is an important part of the QADA® method. According to Matinlassi et al. (2002, 31) the stylebase "helps to shift from the notion of architectural styles towards the ability to reason based on quality attribute-specific models". The goals of having a knowledge base are to (1) make architectural design more routine and more predictable, (2) have a standard set of quality attribute based analysis questions and (3) to tighten the link between design and analysis (Matinlassi et al. 2002, 31).

Stylebase is utilized both in quality-driven architecture modeling and quality-driven analysis of existing architectures. When constructing a new architecture model, an architect searches the knowledge base according to the desired quality assets and selects patterns on that basis. When used for model evaluation, an architect detects which patterns have been used in an architecture model and then checks from the stylebase which quality-attributes are associated with these patterns. (Niemelä et al. 2004.)

As stated previously, recognizing and defining quality attributes is an essential part of the quality-driven architecture development. The quality attributes mentioned in this thesis can be defined as follows:

Maintainability refers to the ease of which software can be modified or adapted to a new environment. Modifications may include building extensions, making changes, and porting to a new operating system. The following are sub-attributes of maintainability. *Extensibility* refers to the ability of the system to acquire new components. *Portability* is defined as the systems ability to run under different computing systems (hardware and/or software). *Modifiability* means the ability to make changes quickly and cost efficiently. (Matinlassi & Niemelä 2003.)

Scalability refers to the ability of software to handle a growing amount of work in a graceful manner or to be readily enlarged. (Bondi 2000.)

Security is system's ability to prevent unauthorized usage while continuing to provide services for authorized users. Unauthorized usage includes unauthorized access, as well as unauthorized operations by legitimate users. (Bass et al. 2004, 85–86.)

Usability is defined as ease with which users can employ software in order to achieve an intended goal. This includes, for example, error handling, *learnability* and efficiency. Learnability means the ease of which users can learn the system's interface while efficiency refers to the amount of time it takes to complete the task at hand. (Bass et al. 2004, 31.)

2.2 Free and Open Source Software Development

Open source software is software which is available with its source code and distributed under a license that allows anyone to use, modify, and distribute the modified or unmodified version of the software (Open Source Initiative 2006). *Free software* is an alternative term to open source software, promoted by the Free Software Foundation. Free software is not the same as *freeware*, software distributed at zero-cost (Fogel 2005, 11). The Free Software Foundation (2006a) describes free software as "a matter of liberty, not price". The definition of free software (Free Software Foundation 2006a) is almost identical to the definition of open source software (Open Source Initiative 2006). The primary differences between the free software and open source software movements are ideological (Vainio & Vaden 2006) and not within the scope of this thesis. In this work, the terms free software and open source software are used synonymously. In many occasions, the acronym FLOSS (Free/Libre/Open Source Software) is used as a neutral term.

Even though there are many different free and open source licenses, all of them say the same thing in the most important respects. Such licenses grant everyone a right to modify the source code and redistribute it either in original or modified form and state that the authors provide absolutely no warranties. Licenses differ in what terms covered software can be used in combination with other software. FLOSS licenses can be grouped into three main categories: Firstly, there are licenses (e.g. the Apache License, the MIT License) that allow any usage, including usage in closed-sources programs. Secondly, there are licenses (e.g. GNU Lesser General Public License, Eclipse Public License) that allow combining the unmodified version of the licensed program with a closed-source program – on the condition that they are clearly separate works. However, these licenses require that any modified version of the licensed work is distributed under the same terms as the original work, i.e. remain open source. Thirdly, there are licenses (most famously GNU General Public License) that do not allow combining the covered source code with closed-sourced programs under any circumstances. These licenses require that any software package, which contains non-trivial amount of licensed code, is distributed under the same terms, with no additional restrictions. GPL-licensed software can be combined with other open source software only if the terms of the two licenses are legally compatible.

As everyone has a right to distribute copies freely, open source programs cannot be traded the same way as proprietary, closed-source programs. Free and open source software is traditionally developed by volunteers. Even though company involvement and consequently the number of paid developers is increasing, the majority of FLOSS developers still work without any monetary compensation (Vainio & Vaden 2006). Many projects are created out of a personal need for a tool and the program is shared in

the hope that someone else with similar needs will expand it and fix its problems (Raymond 2001, 24). People also have other differing motivations of attending, such as self-enjoyment, peer-recognition, gift-giving, and learning (Vainio & Vaden 2006). Raymond (2001, 21) likened open source software development to “a great babbling bazaar of differing agendas and approaches." Such a “bazaar” welcomes people whose motivations, skills, and participation time may vary significantly in working together with no clearly defined roles.

FLOSS community roles are traditionally depicted using an onion model (Figure 4). According to Vainio & Vaden (2006) people start at the outer layer of the onion as *passive users*. Once they begin to explore how the system works, they become *readers.* Some get more involved by *reporting bugs*. Some users may become *developers* who fix bugs and make minor enhancements. A few developers then get more involved and join the ranks of *core developers*, being granted a permission to check-in changes to the source code. At the heart of the community there is occasionally a single person, the *project leader*, who is one of the most active developers. The outer layer of the onion is the largest group, the group size reduces when approaching the core.

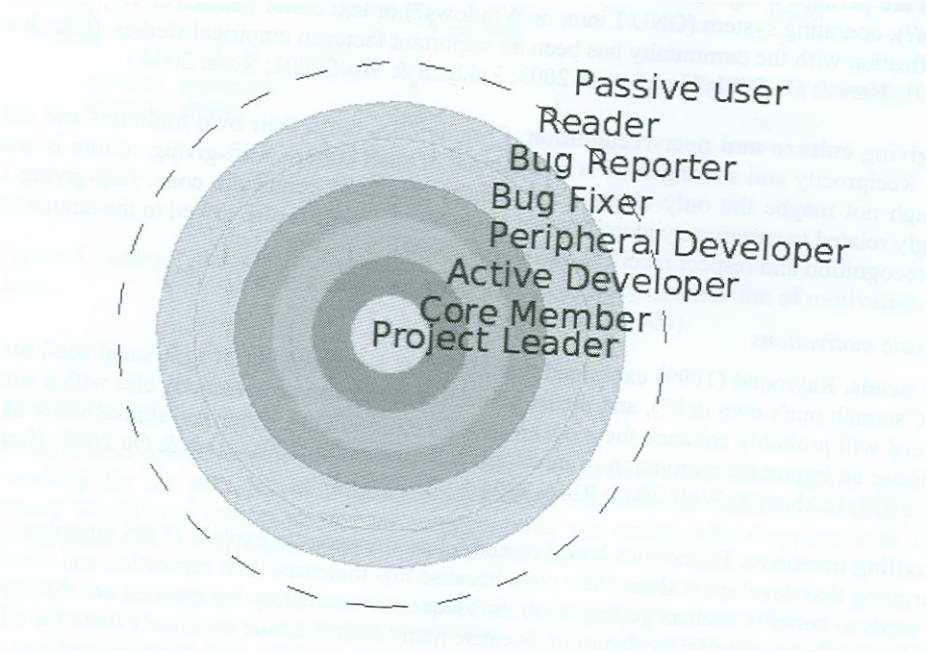

Figure 4. The onion model of FLOSS communities (Vainio & Vaden 2006).

Every successful open source project has some form of governance. In most small and medium sized projects, the authority making the final decision relies on one person who is called the “*benevolent dictator*”. Despite the name given to the role, successful benevolent dictators do not dictate much. It is know that good developers will not stay around for long unless they can have influence on development decisions. Therefore,

the strategy is to let things sort out by themselves through discussion and experimentation whenever possible. As projects mature, the governance model usually moves away from benevolent dictatorship towards a consensus-based democracy. The details on how such systems work vary, but two common elements can be recognized. (1) The community works on consensus most of the time. (2) If a debate does not reach consensus, a democratic decision is made by voting. Inability to reach a decision that everyone can accept can lead to a revolt and cause the project to be *forked*. *Project fork* happens when a developer, or a group of them, take a copy of source code and start to develop a competing product independent of the original project. Forks are bad for all parties because they duplicate the development effort and confuse users on which package to choose. The possibility of forks has significant impact on the governance of open source projects. The more the threat of a fork grows, the more people are ready to compromise. (Fogel 2005, 88–91, 225–226; Goldman & Gabriel 2005, 233–234.)

FLOSS development model differs from its traditional counterpart. While traditional software projects start with a detailed requirements document, open source project usually start with vision and a code artifact which embodies that vision – at least in spirit. Initial requirements are communicated to the community by means of the code artifact. As the project matures, more requirements come in from the community. They may arise from a discussion on a mailing list or be entered by regular users via a requirements logging tool. Most requirements are *post-hoc.* This means that a feature is added when a developer, who wanted the feature, has already provided the coding effort to make it operational. (Scatcchi 2002, 24–38.) Implementation and testing are often going on in parallel with the actual system specification. Individual developers select small parts on which they like to work and they are free to design, implement, and test them as they see fit. There are often competing design and implementations and at most one of which is selected for the project. (Hissam & Weinstock 2005.) Following Raymond's (2001, 28–29) principle "release early, release often" open source products generally undergo a lighter testing process compared to commercial off-the-shelf components. End-users are relied upon in doing some of the testing.

In order to comply with the unconventional development model, the structure of an open source program needs to be modular. Large monolithic software requires too much intellectual investment to learn the architecture, which is daunting to potential contributors and leads to a loss of conceptual integrity. On the contrary, in a well-modularized system, developers can "carve off chunks" and work on them without breaking other parts of source code. (Goldman & Gabriel 2005, 192–193; Hissam & Weinstock 2005.) Marc Fleury (& Lindfors 2001), a founder of the JBoss open-source project, wrote:

> Modularity in open source is not just a good idea, it is the only way a project can mature. Successful open source projects are usually measured by a number of people who participate in the development. […] the key to growing the code base is to enable a lot of developers to work around the core.

Hence, it is not surprising that, when asked to select the most important quality characteristic of the software architecture, almost all open source developers chose modularity (Matinlassi 2007).

Open source developers typically work in a distributed development environment and communicate via Internet. According to Raymond (2001, 34–36), good information management is what prevents FLOSS projects from falling pray to Brooks law (Brooks 1995, 25), the belief that adding manpower to a late software project adds more time to it. Open source projects utilize various technological tools to support the selective capture and integration of information (Fogel 2005, 45). Any healthy open source project applied at least the following five core tools: a website, mailing lists, bug tracker, version control system, and real time chat. (1) Typically, the *project website* serves both as one-way conduit of information to the public and as an administrative interface to other project tools. It should contain documentation for both developers and end-users and, of course, a link for downloading the product. (2) *Mailing lists* are usually the primary communications forum. They also "record" development discussions for future reference and thus form part of project's documentation. (3) A *bug tracking system is* a software application that is designed to help programmers keep track of reported software bugs. It is sometimes regarded as a sort of *issue tracking system*, maintaining and managing a list of various development issues, such as feature and support requests. (4) *Version control system* refers to a combination of technologies for managing multiple revisions of the same unit of information, in this case, source code, web pages, and documentation related to an open source project. The system keeps track of all work and all changes made to project files and enables several developers to collaborate in a distributed environment. (5) Many projects also offer a *real-time chat rooms* using Internet Relay Chat (IRC). In such forums, users and developers can ask each other questions and get replies immediately. (Goldman & Gabriel 2005, 138–147; Fogel 2005; 47–48.)

This section has given introduction to specific aspects of FLOSS development and shown how FLOSS challenges conventional thoughts on software business, software development models, and organizational structures. In the next section, one successful open source project – Eclipse – is introduced.

2.3 Eclipse Framework

Eclipse is known as the most popular open source development environment and a vendor-neutral platform for tools integration. Eclipse community has distinguished in productivity and creativity (Kidane & Gloor 2005) and has developed new features that are evolving Eclipse towards a platform that is integrating not only tools but also applications and services (Gruber 2005). As a result, Eclipse is now more than a simple development environment, it has become a platform that serves the entire application life cycle (Varhol 2006). It can be said that Eclipse form an open eco-system around royalty-free technology. (Henttonen & Matinlassi 2007.)

The architecture of Eclipse has been built from point of view of extensibility and integration. Unlike most other IDE's, Eclipse is what Birsan (2005) calls *pure plug-in architecture*. In Eclipse, there are no core tools in the platform itself. All tools, including the graphical user interface, have been implemented as extensions, a.k.a. plug-ins. The only component that needs to be always loaded is a small kernel called *plug-in loader* a.k.a. *platform runtime*, which takes care of activating the extensions (Hakala 2005). Each plug-in can define its own *extension points* which allow third party plug-ins to enhance the existing plug-in in a controlled but loosely coupled manner (Clayberg & Rubel 2006, 112).

The *Eclipse platform* is a package that includes the plug-in loader and four tools – *workspace*, *workbench, help system* and *team support* – all implemented as plug-ins. (1) Workspace is responsible for managing *resources*, i.e. projects, files, and folders, which the user or tools interact with when using Eclipse. *Projects* are top-level containers which map into user-specified directories in the file system and contain necessary files and folders. Workspace maintains a low-level history of changes to each resource which makes it possible to revert to a previously saved state. It also notifies interested tools about changes to the resources concerned. (2) *Workbench* is the user interface of Eclipse which provides a consistent front-end to the Eclipse tools. In addition to displaying menus and toolbars, it is organized into *views* and *perspectives*. Unlike most Java applications, Eclipse workbench looks and feels like a native application. This is because it is built using *Standard Widget Toolkit* (SWT) – a graphics library that maps directly to operating system's native graphics – and *JFace* – a user interface toolkit build on the top of SWT. SWT needs to be ported into each operating system, which is one of the most debated aspects of Eclipse. As SWT has already been ported to dozens of operating systems, it is not a concern in this study. (3) The Eclipse help component is an extensible documentation system. Mirroring the way plug-ins extend other plug-ins, tools documentation can insert topics into a pre-existing topic tree. (4) Team support is a plug-in that enables the use of version control system for managing users' projects. (Gallardo et al. 2003, 8–10.)

Eclipse Software Development Kit (SDK) consists of the basic platform plus two major tools that are useful for plug-in development. The *Java development tools* (JDT) implement a full featured Java development environment. The *Plug-in Developer Environment (*PDE) adds specialized tools that facilitate the development of plug-ins. Figure 5 illustrates the components of Eclipse SDK.

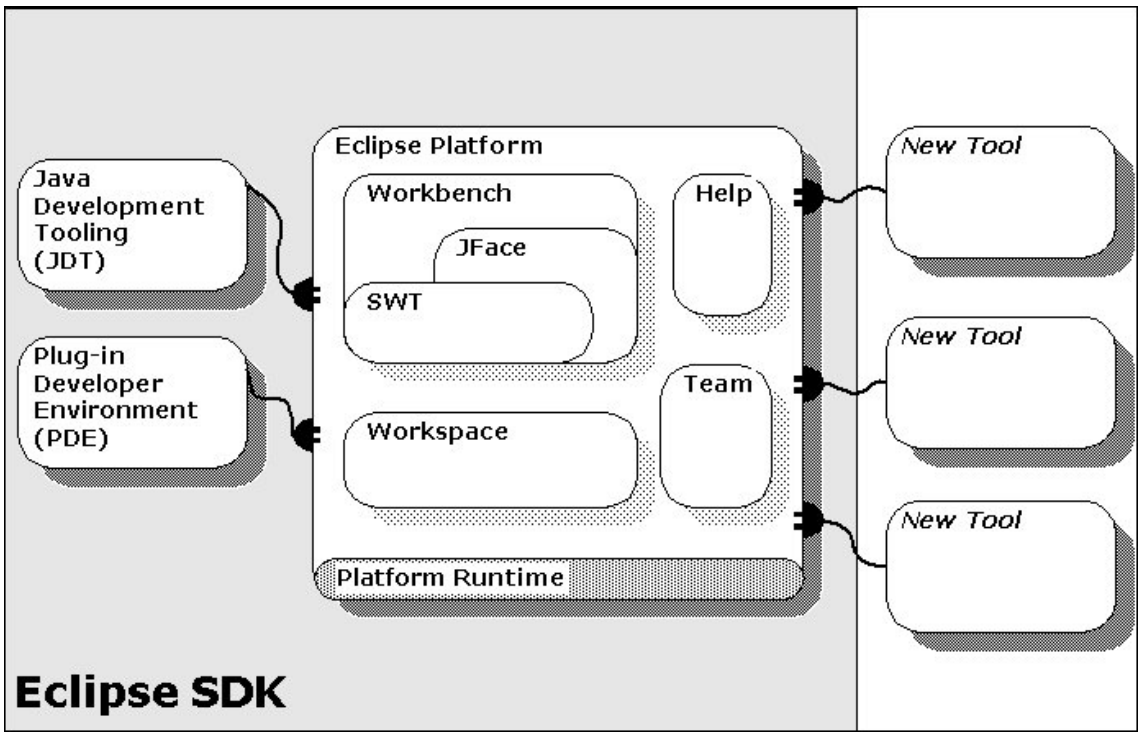

Figure 5. The Eclipse SDK architecture (IBM Corporation and others 2003).

Eclipse community is a group of users, developers, and documentation specialists who share interest in the continuous development of the Eclipse ecosystem. The official Eclipse projects consist of 9 open source projects with more than 50 subprojects (Eclipse Foundation 2006a). In addition, there are hundreds (probably thousands) of Eclipse plug-in projects hosted independently from the official Eclipse project (Clayberg & Rubel 2006, 101). Table 4 summarizes the main stakeholders of the Eclipse community (Beck & Gamma 2004). In this case study, I am acting in three different stakeholder roles: *extender, publisher,* and *enabl*er.

Table 4. Stakeholders in the Eclipse community.

Stakeholder	Description
User	Uses Eclipse as it is.
Configurer	A user who customizes his/her experience of Eclipse within the limits envisioned by the original programmer.
Extender	A programmer who makes extensions by plugging in functionality.
Publisher	An extender who makes extensions available to others for loading them.
Enabler	A publisher who has defined one or more extension points for a plug-in, thus enabling others to extend the contribution.
Committer	Modifies the Eclipse code and incorporates changes into the global Eclipse release. Requires trust of existing committer community.

The development of Eclipse is overseen by the Eclipse Foundation, which is a non-profit entity formed to advance the creation, evolution, promotion, and support of Eclipse. The foundation has full-time management organization that works with commercial developers and consumers, academic and research institutions, standards bodies, tool interoperability groups, and individual developers (Eclipse Foundation 2006a).

3. Development of basecode

As discussed previously (see 1.2), the open source project will be announced when the code artifact has been implemented into a "minimal but working version". This section discusses the development of such a base code for the Stylebase for Eclipse. This does not only mean implementing a certain amount of fixed functionality, but also designing a core which is easy to expand.

3.1 Studying existing pattern tools for re-use

The development phase started by studying the existing pattern management tools. The tools that are open source and have been integrated into Eclipse IDE were tested and compared to each other. The study on existing tools was considered important for two main reasons:

1) To avoid duplicating existing efforts

There are only a limited number of open source developers interested in the given area, and it is hard enough to create one community of them, let alone two. One of the key points of the open source philosophy is to avoid having to reinvent the wheel. If there was already an active community with goals similar to ours, I should join their endeavors. (Goldman & Gabriel 2005, 251; Fogel 2005, 19.)

2) To find re-usable code

According to Robbins (2005, 251), open source projects which re-use existing code efficiently tend to be more successful. This is because they can demonstrate results sooner, focus on the added-value, and adhere to the community's culture of re-use (Robbins 2005, 251). It was thought that, even if a project was not worth joining, time could be saved by copying some of the source code.

From these two perspectives, I started looking at the tools listed under category "pattern" on the Eclipse plug-ins homepage (www.eclipseplugins.info). Other sources of information, Freshmeat and SourceForge websites, for example, were also used. Seven open source pattern management plug-ins for Eclipse were found. Even though two of them had no information on license, they were probably intended to be open source.

Comparison of the tools was time-consuming due to the lack of documentation. Website provided some feature lists, but there was often no clear distinction between

implemented and planned features. In practice, the only way to check a plug-in was to download it, play with it, and try to figure out how it works. Appendix 1 provides the full list of the examined plug-ins, their primary features, as well as their advances and disadvantages. PSE (Pattern support for Eclipse) appeared the only interesting tool for our purposes. Despite not being compatible with any UML modeling tool, it contains a considerable amount of potentially re-usable code. WebOfPatterns could become interesting if the promised feature of browsing online pattern repositories is implemented. SEDS plug-in should be evaluated as soon as its user interface has been translated into English.

As PSE had not been published under GPL or any other OSI-approved open source license, it was not clear if its source code could be used. The University of Linz, the holder of the tool's copyright, was contacted and I received permission to use PSE's source code in Stylebase project as I wish (Sametinger 15th May 2006, e-mail message). Some commercial Eclipse extensions, for example, that of Borland (Borland Software Corporation 2006), are known to include pattern management tools. These were not evaluated in detail, because closed-source projects do not compete for the same developer resources and do not provide a possibility of code re-use.

3.2 Requirement specification

In FLOSS development model, requirements engineering process is user-driven and decentralized (see section 2.2) and therefore the requirements are most likely to increase beyond those foreseen. The core of the product should be designed so that the growing requirements lists does not lead into a major "feature creep", a situation were implementing originally unplanned functionality lowers the quality of the product (Robins 2005, 250). In addition to identifying the requirements for the initial code artifact, it is essential to identify the requirements that enable the product to evolve according to the needs of the user community. The requirement specification of the Stylebase for Eclipse has been divided into two subsections. The respective sections define (1) the functional and directive quality requirements that need to be implemented before, or at least very soon after, the project is made public and (2) functionality and quality characteristics which prepare the product for continuous evolution.

3.2.1 Initial functional requirements

Initial functional requirements of the Stylebase for Eclipse were gathered by exploring the functionality of the stylebase tool implemented in 2004 (see section 1) and interviewing VTT personal (Niemelä 15 May 2006; Matinlassi 18 May and 4 September

2006; Merilinna 20 May and 3 August 2006; Tarvainen 20 September 2006, interviews). The most essential functional and directive quality requirements are summarized in Table 5. These features must all be implemented before the project was to be made public. For a more detailed requirement specification, which also includes requirements implemented by external contributors, please see Appendix 2.

Table 5. Summary of essential functional requirements.

Functional Requirement	**Directive Quality Requirements**
● Store architecture and design patterns plus associated quality attributes in DBMS. ● For each pattern, store name, description, data model (xml), diagram picture and instructions for usage (html)	Extensibility: ● Support to store macro, micro and reference architectures (plus other so for undefined model types) ● Support to store the data model in any (textual) form desired
● Provide a database interface for accessing the pattern repository and managing its data	Scalability: ● Support for several different tools and several users per tool to access the data simultaneously Security: ● User authentication with read/write permissions
● Provide GUI for browsing, searching and updating patterns ● Include searching by quality attributes and a free text search on the pattern description	Usability: ● Fast and easy browsing for end user satisfaction Scalability: ● Independence of any particular UML modeling tool
Provide GUI for exporting/importing pattern files to/from DBMS	Security/Usability: ● Locking to prevent users from overwriting each others changes

3.2.2 Encouraging evolution

Table 1 (see Section 1.1.5) presented the hierarchical stack of stakeholders of the Eclipse community. The stack is built from a technical point of view. That is, publishing an Eclipse extension does not require making it extendable. Although this is true, attracting an active open source community, as stated previously (see Section 1.1.4), requires a modular architecture, which in this case includes the efficient use of Eclipse extension mechanisms.

Stylebase for Eclipse must implement modularity on two levels:

1) The internal architecture of the Stylebase for Eclipse must be modular. The core application is split into independent modules by following a commonly-accepted software architectural pattern.

(2) The Stylebase for Eclipse must provide functionality which lets users build custom extensions without touching the source code of the Stylebase for Eclipse. To achieve

this, Stylebase for Eclipse provides both *access points* and *extension points* for dependent plug-ins to use.

Access points define a set of functions which developers of third-party plug-ins may use without detailed understanding of their internal workings. Access points form an application programming interface (API) of Stylebase for Eclipse. Extension points let users, not only to use, but also enhance the functionality of the main plug-in.

3.3 Database design

Stylebase for Eclipse is based on a database management system (DBMS). MySQL was selected over other open source DBMS (e.g. PostgreSQL, Firebird and Ingress) because it is a most widely used one. The popularity means better tool support, more resources, and a larger community to help with technical problems (Gebert 2003). Despite not fully complying with the ANSI standard, the latest version supports all commonly used SQL-92 features, for example, sub-selects, word indexes, transactions, views, and stored procedures (MySQL Ab 2005, 4–12).

Maintainability, not performance was the main consideration when designing the database schema. This is because as the product is likely to manage mostly small or medium sized databases. In order to keep data model simple and easy to maintain, the database was designed in *the third normal form* (3NF). In the relational database theory, normalization is the process of reconstructing the logical schema of a database in order to eliminate redundancy, improve efficiency and prevent errors during data operations. The third normal form is a normalization level which requires that (1) values in each column of a table are atomic (2) all columns of a table depend directly on the primary key. (Halpin 2001, 627–633.)

Three database tables are required to store the contents of the stylebase. The tables are summarized in Table 6.

Table 6. Database tables summarized.

Table name	Description
Patterns	The table contains name and properties of each pattern. Most importantly it contains large object fields for storing *guide*, *picture,* and *model*.
Quality Attributes	The table contains information on which patterns are associated with which quality attributes, as well as respective rationales
Attribute Definitions	The table contains names and definitions of all quality attributes used

Guide is the documentation of a pattern, typically stored in HTML format. It includes application areas, instructions for usage, plus textual descriptions on the role of each component. *Picture* is graphical representation of the pattern in binary format (e.g. JPEG, GIF). In this context, *model* is the data model of a pattern, i.e. structural representation of its components and their interrelations. Typically, the text field contains an UML diagram in XML format (see section 2.1.1).

Figure 6 presents the most essential fields of the tables and illustrates dependencies between them. The fields that form the primary key of a table are underlined. The abbreviations "PK", "U", and "I" stand for primary key, unique index and index (non-unique) respectively.

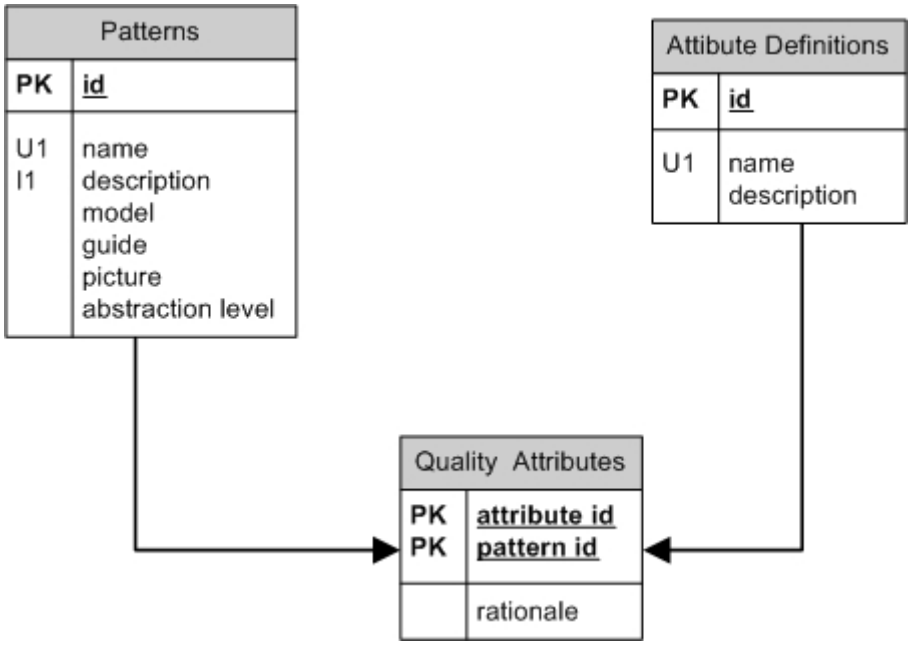

Figure 6. Database Schema.

For more detailed description of the database schema, see Appendix 3.

3.4 Plug-in architecture

It was decided that the architecture of the Stylebase for Eclipse should apply the well-known model-view-controller architectural pattern (see e.g. Bushmann et al. 1996, 125). The MVC architecture is a way of breaking an application into three parts: model, view, and controller. The user input, the manipulation of data, and the visual feedback to the user are separated and handled by controller, model, and view objects respectively.

MVC architecture was selected for the following reasons (Henttonen & Matinlassi 2007):

1) It promotes extensibility by allowing multiple representations (views) of the same information (model). This makes it easy to update the graphical user interface, which is especially prone to change requests, and/or customize views based on user's profile.

2) It promotes code re-use by allowing the same view to show data from different models. By adding a new model, one could adjust the tool to entirely manage different types of data with minimum recoding effort.

3) It eases maintenance by allowing an individual developer to focus on one aspect of the application at a time. Multiple developers can simultaneously update the interface, logic, or input of an application without affecting other parts of the source code.

4) MVC architecture is well suited for Eclipse plug-in development, because the view classes of Eclipse can receive any other class as an input object. Eclipse platform itself has a model-view-controller architecture (Griffin 2004).

The QADA method (see section 2.1.2) defines two abstraction levels for software architecture modeling – conceptual and concrete. This section describes the software architecture on conceptual level from three different viewpoints: structural, behavioral, and deployment. UML elements have to be used in a non-standard way in order to present some of the viewpoints as defined by the QADA method. The concrete level architecture description can be found in Appendix 4.

Structural View

Figure 7 shows how the plug-in implements a model-view-controller pattern. In order to increase the level of modularity, the three main components communicate with each other via predefined interfaces.

The *view* is responsible for providing a graphical user interface. A *controller* is responsible for mapping GUI events to application response. A *model* contains the core functionality of the application. It is divided into two subcomponents: *container* holds pattern data during programs execution and *admin* encapsulates methods that manipulate it. Model attaches to *database* for which servers are a permanent storage for the pattern data.

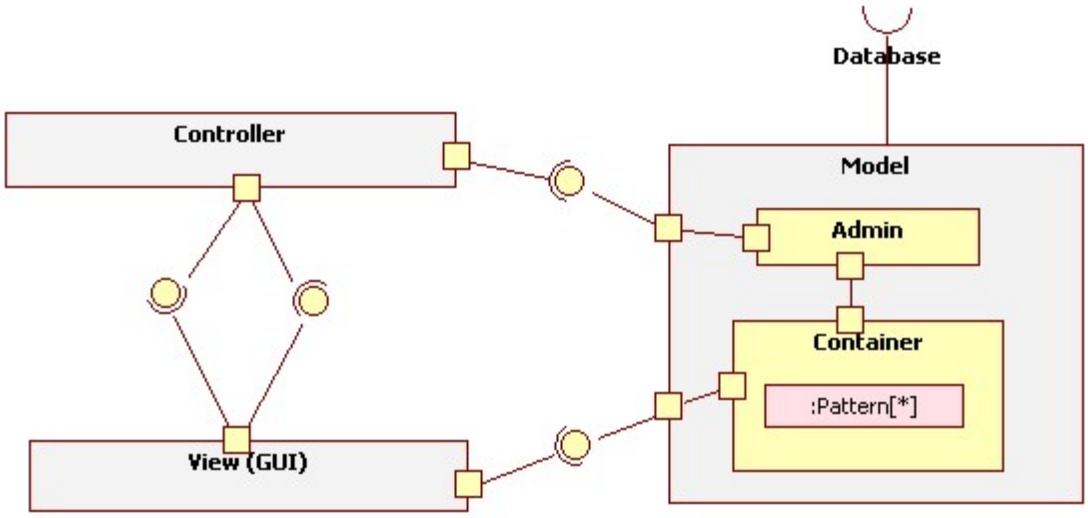

Figure 7. The Stylebase for Eclipse – Structural View.

The requirement specification determines (see Section 3.1.2) that Stylebase for Eclipse must provide access and extension points for downstream plug-ins. They are described in Table 7 and Table 8 respectively.

Table 7. The access points provided by the Stylebase for Eclipse.

Controller	The interface provides access to the control component. It lets users associate the functions of Stylebase for Eclipse with the GUI of another plug-in.
Model	The interface gives access to the model component. It provides a set of methods for retrieving and updating essential data in the Stylebase.
Database (SQL)	The interface provides SQL-level access to the underlying database. It helps in implementing specific functionality not provided by the model interface.

Table 8. The extension points provided by the Stylebase for Eclipse.

Model Extension Point	The extension point provides the means of adding new models, units for storing, and handling different types of data.
GUI Extension Point	The extension point provides the means of customizing the user interface of the Stylebase for Eclipse. It allows users to add their own views and/or menu items to the main view of the Stylebase for Eclipse. The controller component is extended respectively.

Behavioral View

Figure 8 illustrates the communication between the main components.

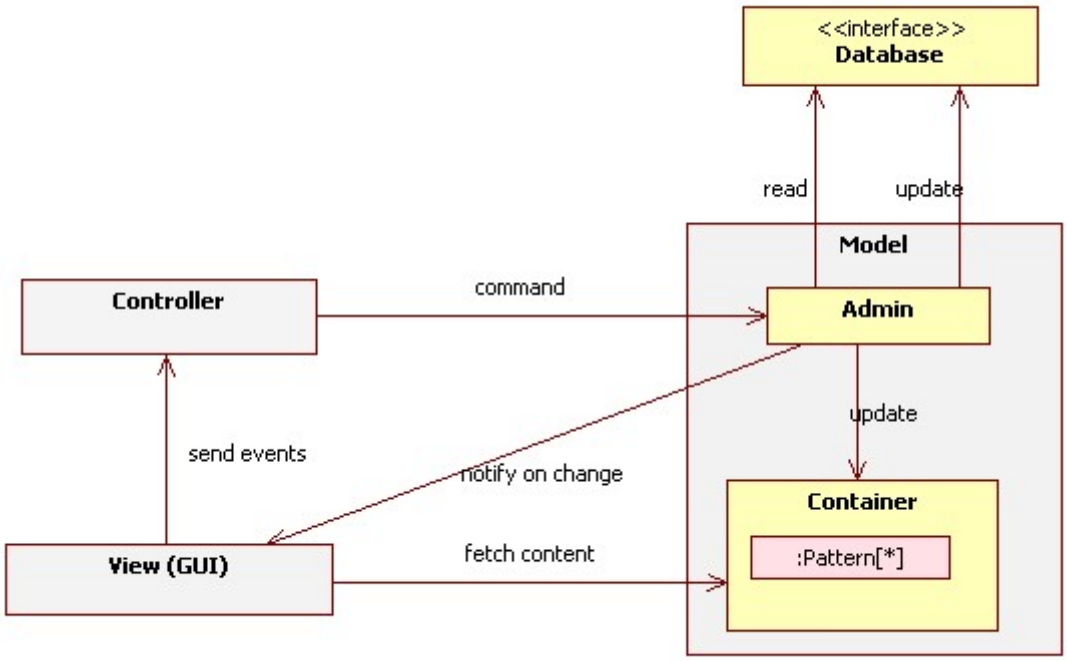

Figure 8.The Stylebase for Eclipse – Behavioral View.

The *view* component attaches to a model and shows its contents on the display. The model notifies the view when its contents have changed and then the view redraws the affected part of the image to reflect these changes. The view detects GUI events (e.g. mouse click, button press) and sends them to the controller. A *controller* receives events from the view and then instructs the admin part of the model(s) to perform actions based on the input. The model admin updates data both in the model container and the remote database. Upon initialization of the program, the model admin reads data from database and fills the container.

Deployment View

Figure 9 illustrates how the Stylebase for Eclipse is deployed.

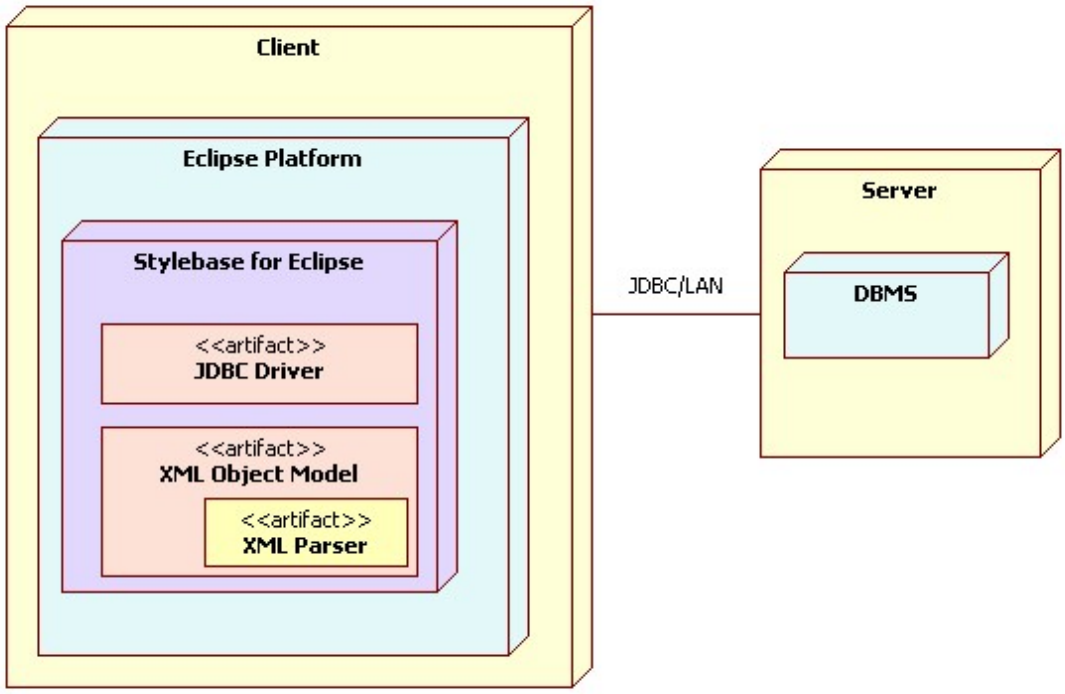

Figure 9. The Stylebase for Eclipse – Deployment View.

The Stylebase for Eclipse is an extension to the Eclipse IDE (see section 2.3) which is always deployed on the client machine. The Stylebase for Eclipse includes two third-party subcomponents: JDBC Driver and an XML Object Model. JDBC Driver is a client-side program which sends and receives requests to/from DBMS. An XML object model is a programming interface for processing XML Documents with Java. It includes a separate component which does the actual parsing. Database server is typically deployed on server machines. If desired, one computer can act both as a server and a client, in which case, all applications are installed on the same physical machine. In any case, a client-server type connection is established between the database server and the client libraries.

Development View

The Stylebase for Eclipse project is by no means a unique case – it is dependent on various tools developed by other open source communities. Figure 10 shows the selected technologies and their providers.

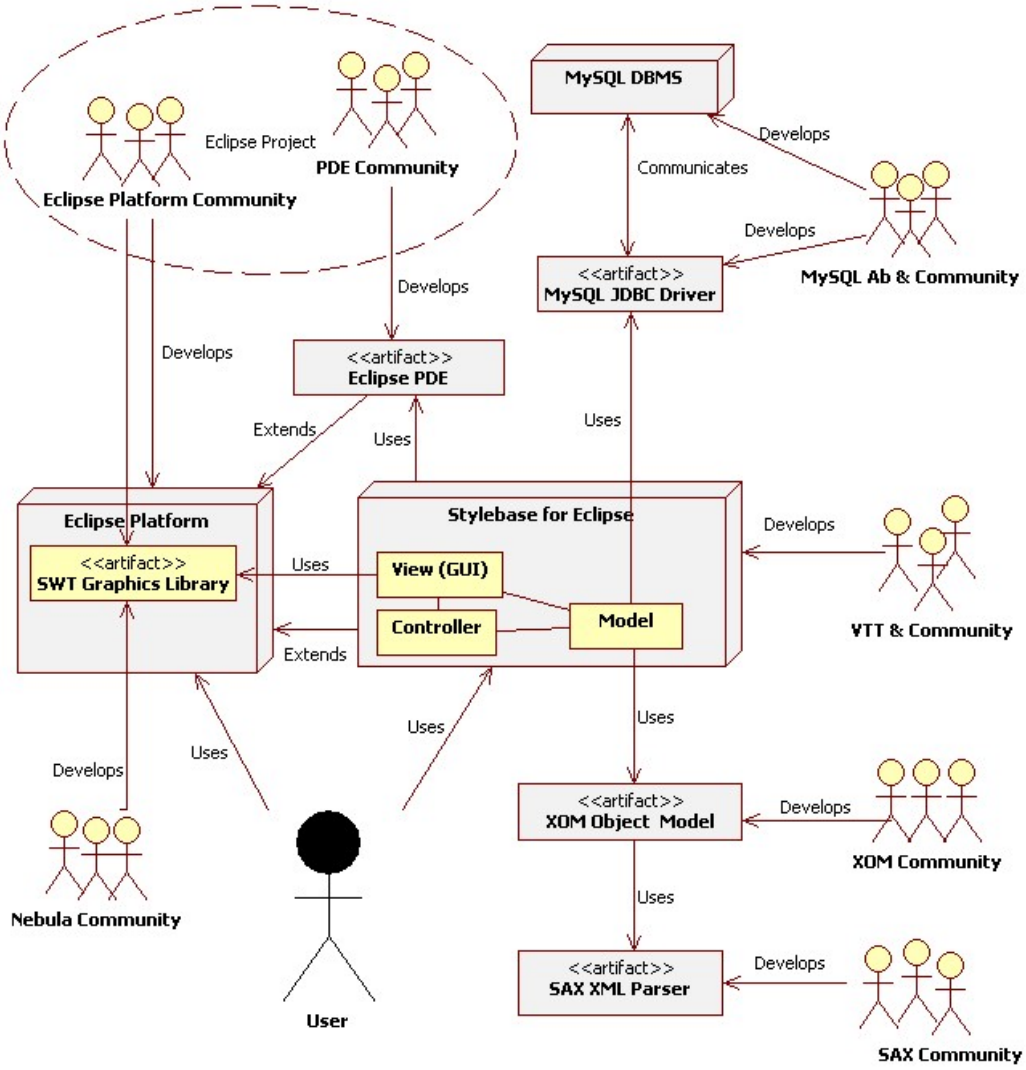

Figure 10. The Stylebase for Eclipse – Development View.

MySQL DBMS (see section 4.3) and the associated JDBC Driver, namely MySQL J/Connector, are provided by the company called MySQL Ab and the FLOSS community it supports. XOM was selected as an XML object model mostly for licensing choices as shall be explained later in the document. Eclipse Platform (see section 2.4) and Eclipse Plug-in Development Environment (PDE) are developed by respective communities under the official Eclipse project. The development of Standard Widgets Toolkit (see section 2.4) is managed by the Eclipse platform community, but new widgets originate from the Nebula project which is a source of additional SWT widgets and an "incubator" for SWT.

4. Community founding

4.1 Choosing and applying license

Many factors have an effect on the licensing policy of an open source project (see section 2.4). Both Fogel (2005, 33) and Goldman and Gabriel (2005, 190) state that it is best to choose one of the well-known existing licenses, rather trying to create a new one. A new license, even if short and clearly written, is an additional hurdle that limits outside participation. I decided to use the GNU General Public License (GPL), which has been written by the Free Software Foundation. Many contributors are familiar with it as it is the most widely used open source license, which is, according to Fogel (2005, 34) a big advantage on its own. GPL ensures that the Stylebase plug-in can be unambiguously combined with MySQL client libraries which are also GPL licensed. As GPL prohibits distributing the project source code as part of a closed-source product, it encourages contributing extensions and improvements back to the community (Fogel 2005, 237; Godlman & Gabriel 2005, 123–125). The main drawback is that the terms of the GPL may put companies off contributing to a project (Goldman & Gabriel 2005, 190). On the other hand, one should note that GPL does not prevent a company from using the product as part of a closed in-house system – it only impacts software vendors who intend to sell a derived work (Rowan 2006). Another disadvantage of GPL is the incompatibility with Eclipse Public License (EPL) (Free Software Foundation 2006c). There are expectations that the issue may be solved by GPL version 3 (O'Riordan 2006).

Merely stating that the program is distributed under a certain license is not sufficient for legal purposes. For that, the software itself must contain the license. The normative way to do this is to place the full license text in the file called COPYING (or LICENCE) and then put a short notice on top of each source file which states the copyright holder, the license, and the location of the full license text. In addition, a README file with the copyright and licensing statements may be placed into each subdirectory containing binary files.

The following text (Free Software Foundation 2006d) was added into respective source and README files of the Stylebase for Eclipse:

```
Copyright(C) 2006 VTT Technical Research Centre of Finland

This file is part of Stylebase for Eclipse.

Stylebase for Eclipse is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.
```

```
Stylebase for Eclipse is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License more details.

You should find full license text in the file called COPYING. If
you did not receive a copy of the GNU General Public License
along with this program, write to the Free Software Foundation,
Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
```

It is notable that, as external libraries distributed with the program have not been modified, their copyright remains intact. This is recognized by mentioning the copyright holders of third-party libraries in the primary README file as well as in respective sub directories.

Once GPL has been applied to the program, its terms of redistribution are *contagious* – they get passed into anything else that code is incorporated into (Fogel 2005, 237). It was thus essential to examine that all third-party files distributed with the Stylebase plug-in, including icons, have been published under a GPL compatible license. The Free Software Foundation maintains a useful list showing which free software licenses they consider as GPL-compatible (Free Software Foundation 2006b). Initially, the Stylebase for Eclipse tool was used as an XML object model called JDOM. It was published under its own open source license specifically written for the JDOM project. Due to the fact that JDOM uses a permissive license similar to the MIT (Massachusetts Institute of Technology) license, I considered it to be GPL-compatible. Later, I found out that the clause that prohibits the use of the name JDOM in derivative works can be seen as an additional restriction and this would rule out the GPL-compatibility (Rowan 2006). A polite posting (Henttonen, 28 September 2006, e-mail message) to an open source forum regarding the GPL-compatibility of JDOM caused an intense “flame war”. It was decided to switch to another XML object model which was published under GNU Lesser General Public License – a license which can unambiguously combined with GPL.

4.2 Building infrastructure for information management

Every open source project needs technical tools to facilitate distributed development and encourage information sharing (see section 2.2). According to Fogel (2005, 45), the success of an open source project is dependent on the skill of the project leaders in using these tools and persuading others to use them. An easiest way to set up the required tooling environment is to subscribe to a *canned hosting site*, i.e. a server that offers prepackaged web areas with all tools needed to run an open source project. The free hosting services provide e.g. web hosting, mailing lists, revision control, and issue trackers (see section 2.2). Using a canned site saves a lot of time and effort, because the

provider takes care of selecting and configuring the tools, as well as maintaining them, e.g. taking backups of the data stored in the tools. The other advantage of canned sites is that they offer huge amount of bandwidth and server capacity, making sure that unexpected peaks in the number of visitors can hardly bring the site down. On the other hand, the canned sites are adjustable only with certain parameters and thus setting up a project site by oneself gives more freedom. Considering the limited resources and the laborious effort of self-hosting a project, it was decided to use one of the canned hosting sites. (Fogel 2005, 84–85.)

For selecting a hosting facility, the most well-known, general-purpose hosting facilities – SourceForge.net, Savannah, BerliOS – were evaluated by using an evaluation framework created by Dr. So (2005). The all evaluated facilities are so called *infrastructure sites*, this means that most information about the projects hosted is stored in databases and a user interface is provided for setting up a project. They all run a collaborative software development management system originally called "SourceForge". SourceForge.net uses a closed-source proprietary version of this software, while the two other websites run free versions forked from the last open source version of SourceForge. They all essentially provide the same range of tools and services, BerliOS slightly falling behind the two others. Savannah provided by the Free Software Foundation was selected as a hosting provider because it is ad-free, provides the most professional looking user-interface, and monitors the quality of the projects hosted.

Savannah is dedicated to free software advocacy and has strict hosting policies, such as code review before accepting a new project as a Savannah project. The Savannah administration requires that all hosted software 1) is licensed under a free software license compatible with the GNU General Public License, 2) runs on at least one free operating system such as Linux, and 3) is not dependent on any non-free software. Non-free software includes all proprietary programs (such as Macromedia Flash and the Microsoft SQL Server), non-free file formats (such as the Graphic Interchange Format), and non-free programming languages (such as the Sun implementation of Java). (Free Software Foundation 2006c.) Savannah administration also imposes strict requirements on how a free software license should be applied to the code artifact, including external libraries. The purpose is to ensure that all code distributed at Savannah is "legally sound" (Robson 12th December 2006, e-mail message). In the case of the Stylebase plug-in, this meant a lot of additional work. For example, one needed to open the JAR (**J**ava **Ar**chive) file of the MySQL client, place a README file into dozens of sub directories, and then to package the source code and binaries back into a JAR file. Savannah also requires that the source code of all external libraries is distributed with the plug-in. It is not sufficient to provide instructions on where to download the source code (Robson 12 October 2006, e-mail message).

The application process to Savannah took longer than expected. Due to the strict deadline of publishing the project, it was eventually decided to subscribe to SourceForge.net instead of Savannah. SourceForge.net has a semi-automatic acceptance process and therefore our application was processed in less than a day. The main disadvantages of SourceForge.net are online ads, although relatively non-disturbing, requirement to place SourceForge banner on each hosted webpage and the difficulty to "stand out" from over hundred thousand miscellaneous projects hosted. On the other hand, SourceForge.net is the most well-known service – which can be seen as an advantage – and also provides the best tools for collecting and analyzing project statistics.

The following tools were configured at SorceForge.net: a version control system, two mailing lists – one for users and another for development discussions – and three issue trackers – for bug reports, support issues, and feature requests. The project website at http://stylebase.sourceforge.net is a portal to all aspects of the project. It provides access to the project tools and documentation and naturally a page where users can download the product and its source code. The name of an IRC channel, which can be visited to meet developers, is also mentioned at the website.

In addition to end-user documentation required for any software product, open source projects needs clear development documentation. If the internal structure of a program is well-documented, it is easier for new developers to learn their way around the source code. According to Goldman & Gabriel (2005, 139), the easier it is to learn enough to get started, the more developers a project will attract. The design documentation has to be kept up-to-date and, ideally, external contributors should document their own code (Goldman & Gabriel 2005, 140). To facilitate distributed maintenance of development documentation, a wiki was set up for the project. A wiki is a website that allows any visitor to edit or extend its content. Development documentation of the Stylebase for Eclipse was placed at http://eclipse-wiki.info, a wiki server dedicated to hosting wiki pages of Eclipse related projects.

4.3 Announcing and publicity

Marketing is an essential part of founding a new open source community (see section 1.2.4). In order to make the project known to the public, it was decided to publish announcements on some popular, open source related websites and to send one-time postings to carefully selected mailing lists. I chose not to send email to several large mailing lists because such postings are easily considered as spam. As well as regular internet users, open source developers dislike irrelevant information that is blocking their communication channels (Raymond & Moen 2006). Instead, I posted to mailing

lists where our project should be clearly topical and of interest, e.g. lists targeted at developers of Eclipse-compatible modeling tools. The Eclipse Plug-in Central (EPIC) is obviously a website where I wanted the plug-in to be listed. This is a place where Eclipse users search for commercial and open source plug-ins. An announcement was also placed at a few open source development portals, at FreshMeat.net and Tigris.org, for example. Tigris.org is an open source software development community focused on development tools. Freshmeat.net, which Fogel (2005, 43) mentions as the number one place to be seen, is a very popular website that allows people to keep track of the latest FLOSS releases and updates. Furthermore, I promoted the project at various open source related events, such as the Open Mind conference. Open Mind (www.openmind.fi) is the leading annual international open source conference in Finland, covering both business and technological sides of the matter. A brochure and poster were designed for marketing the project. I also contacted lecturers of software development courses at some universities and offered the Stylebase for Eclipse project as a practical work for their students.

Goldman & Gabriel (2005, 246) stresses the importance of establishing strong, positive connections with other communities, especially closely-related ones. The Stylebase for Eclipse should be seen part of the Eclipse community, not as an isolated plug-in project. In order to establish credibility with other members of the Eclipse crowd, it was decided to apply for membership of the Eclipse Foundation. Associate membership is free of charge for non-commercial entities like universities and research institutes. The Eclipse Foundation requires that associate members commit to 1) deliver added value to the Eclipse community within 12 months and 2) publicly announce joining the Eclipse Foundation within 90 days (Eclipse Foundation 2006b). While the associate membership does not grant decisive power, it grants rights to participate in project reviews, project creation, and discussions on Eclipse intellectual property policy. It also entitles the member to use the Eclipse Foundation Member logo in marketing activities (Eclipse Foundation 2006c). VTT published a press release on joining an Eclipse Foundation and it was also used as a means of advertising the plug-in.

Table 9 summarizes some of the activities which assisted in making the Stylebase for Eclipse project better known. The list is by no means complete, but gives an idea of different channels used in promoting the project.

Table 9. Summery of the most important marketing activities.

Week	Activity
41/06	Stylebase for Eclipse was mentioned as part of a lecture on software architectures at the University of Oulu.
42/06	Lectures of software engineering courses at the University of Oulu and the Oulu University of Applied Sciences were informed about the project.
	The Stylebase for Eclipse was mentioned as part of a lecture on Quality-Driven Driven Model-Transformation at VTT.
43/06	One-time postings were made to a few selected mailing lists
	The project was announced at Eclipse Plug-in Central (eclipseplugincentral.com) and Tigris (www.tigris.org)
44/06	Tool demonstrations were delivered and brochures were handed out at VTT booth at the Open Mind event.
	Another research project on open source software (OSSI) was contacted and co-operation was suggested in the form of using the tool and giving feedback
47/06	VTT joined the Eclipse Foundation and announced it in a press release which also mentioned the Stylebase for Eclipse project.
48/06	A presentation on the Stylebase for Eclipse project was delivered in a seminar of an international research project in Malaga, Spain.
	Project was advertised to the representatives of the Fatih University in Turkey. Co-operation is planned.
49/06	A presentation on the Stylebase for Eclipse was delivered in a public seminar held at VTT. Brochures were handed out.
51/06	Project was announced at FreshMeat.net.
	Project was advertised to the representatives of the University of Linz in Austria.
	An article on the Stylebase for Eclipse was published at the website of FSFE (Fellowship of Free Software Foundation Europe).
08/07	The Stylebase for Eclipse tool was demonstrated to some staff members of the engineering faculty of Fatih University in Istanbul.
11/07	The release of Beta 1.0 was announced at the internet forums where the project had been previously advertised (e.g. Freshmeat.net and Eclipse-plug-in central).
12/07	A lecture on the Stylebase for Eclipse project was delivered to engineering students at the Fatih University. A poster was made for advertising the lecture.

4.4 Feedback and contributions from community

Users of the Stylebase for Eclipse send feedback mostly by sending e-mail directly to the project administrator or entering comments into the support issue tracker. It seemed difficult to direct conversations to the mailing lists instead. Some users commented and rated the plug-in at Eclipse Plug-in Central (http://www.eclipseplugincentral.com). By the time of writing, 14 users have contacted us, which is over 10% of the number of downloads made. Some users have only reported one issue while others committed to longer conversations on the development plans of the plug-in.

Generally speaking, it seems that many users are responsive to the idea of maintaining a knowledgebase of architectural models. Users stated for example as follows.

> Thanks for this marvelous plug-in! I was looking for such a
>
> plug-in for quite a while.
>
> The idea [of having a knowledge base] is excellent!
>
> Cool project! I am thinking that I can use this for my own projects and possibly work since I am doing….
>
> I love the idea of the Stylebase plug-in!

Essentially all of the negative feedback concerned the installation process of the plug-in. Most users wanted to keep a small knowledge base for themselves – rather than share one with a large development team – and therefore installing a database system brought no benefits for time. Installation process was made even more laborious by the fact that many were unfamiliar with MySQL. A few excerpts from user feedback are provided below.

> You assume that the installer has a strong understanding of mysql. Bad idea.
>
> The installation process is too time-consuming and error-prone.
>
> How can you justify using a mysql database? Loading each pattern into the database involves too much work. […] Please simplify the installation and configuration process.
>
> How about supporting two persistence models. The default would be a simple local file collection of stylebase models. The other persistence model would use a MySQL database. The simple local file based model would allow export/import so that the models could be shared by sneaker net, email or CVS.

As part of the marketing activities, university lecturers of software engineering courses where contacted and the development of the Stylebase plug-in was offered as practical work for their students. As a result, two students from the Oulu University of Applied Sciences contributed to the development. During the period of six months, the students used a few hours each week implementing new features to the plug-in. Primary contributions were the following: context-sensitive help, GUI for managing various configuration parameters and GUI for modifying quality attribute definitions. The contributions improved usability and simplified the configuration process. Put together, the contributions presented approximately 100 hours of productive programming work, not counting time spent on learning.

5. Discussion

This thesis work aimed at accumulating knowledge in Eclipse and open source software development by means of a "hands-on" experiment. It had three concrete, measure objectives (see section 1.4) which were labeled as technical, business and organizational. The first subsection summarizes some interesting experiences and discusses the lessons learned and the questions that arose. In the next subsection, the achievement of concrete objectives is assessed. After that, near-future development work and research interests are viewed. The section is closed by some notes on the thesis process.

5.1 Experiences and lessons learned

The technical implementation of the plug-in was relatively straightforward for an experienced Java programmer. Naturally, learning to use SWT graphics library (see Section 2.3) and getting familiar with Eclipse programming interface took time. The most interesting experiences were related to founding and managing an open source community. This section discusses experiences on code re-use, selection of hosting facilities, marketing activities, and parallel development.

Even though the reviewed literature proclaimed the benefits of code re-use (see section 3.1), the matter proved to be complex in practice. Some time was initially saved by copying code from another plug-in project, but this soon backfired as a lot of time was spend in harmonizing code styles and debugging the foreign part of the code base. It seems that – unless the amount of re-usable code is immense – there are many good reasons for writing ones own code from scratch. However, studying source code of the other plug-in was very useful for learning to develop Eclipse plug-ins. It seems that, as also stated by Souza (2003), the most important benefit of code reuse is knowledge reuse.

Savannah was first selected as a hosting facility, but the project was eventually set up at SourceForge (see section 4.2). One of the experienced advantages of Savannah was that it would host only carefully reviewed projects – causing the disadvantage of long drawn-out acceptance process. The process was slow most probably due to the following reasons. Not all the requirements for application were clearly stated at the same time, but rather mentioned one after another on a weekly basis. The voluntary nature (Vainio & Vaden 2006, 13) of open source communities was really put into action when our emails were replied to mostly on weekends and there was often a long delay before receiving a reply. In comparison, the application to SourceForge was processed in less than a day. Considering the amount of bogus projects found at

SourceForge.net, the quick approval appears to reflect a loose hosting policy rather than efficiency. (Henttonen & Matinlassi 2007)

It became necessary to replace a subcomponent with another one because the initially chosen component was published under a non-standard license whose legal compatibility with GPL was questionable (see section 4.1). The incident highlighted the risks of using source code from projects which use self-written licenses. It seems that whenever a project uses a non-standard license, compatibility issues are a matter of debate. Problems with incompatibility of GPL with other FLOSS licenses also raised questions on the selected licensing scheme. A non-contagious license (see section 4.1) would cause less incompatibility problems. GNU Lesser General Public License (LGPL) and Common Public License (CPL) are among possible alternatives. Such licenses would allow software vendors to sell Stylebase for Eclipse as part of a closed-source package under certain conditions. It does not seem like such closed-source distribution could harm the project, however, the issue is still to be studied further.

SourceForge platform provided all tools needed and, generally speaking, it was easy to set them up and running. It was possible to customize the tools to meet with the specific needs of the Stylebase for Eclipse project. However, the value of statistics gathered by SourceForge was diminished by the fact that the most interesting data is only archived for less than a month. The improved statistics collection good be a reason to self-host a project, if required resources were available.

SourceForge.net does not work as a marketing channel. Nobody visited the project at SourceForge before I actively started marketing. Dozens of new projects, ranging from games to hardware drivers are added to SourceForge every day and it is obviously difficult to stand out from such a large, heterogeneous crowd. In comparison, much valuable feedback came in as a response to the announcement at Tigris.org (see section 4.3). As Tigris also provides hosting services, it might have been a better choice as a hosting provider. Instead of hosting a huge amount of miscellaneous projects, Tigris focuses on development tools and is visited by the target audience I was looking for. It is still to be studied whether switching the project hosting to Tigris would be worth the effort.

The most effective marketing activity seemed to be giving a demonstration at a seminar targeted for a special audience interested in open source. A concrete metric set for assessing the effectiveness of marketing activities was provided by the number of downloads at stylebase.sourceforge.net. A few days after the demonstration I was able to witness nearly 50 downloads from different IP addresses. Announcement at websites FreshMeat.net and EclipsePluginCentral.com did not bring significant amount of visitors to the site. The measurement was easy as both sites keep record on URL hits.

Almost all traffic occurred while the respective announcement was less than 5 days old, after that the number of hits fell dramatically.

The external developers got started with the development after relatively little time spent it learning. This tells a positive message on the learnability of the selected architectural style and the clarity of development documents. Some gaps in the documentation triggered e-mail discussions – which now complete serve as a reference for new contributors. However, the architecture based on model-view-controller pattern did not support parallel development in the most effective way possible. Most new features would require a developer to enhance all three components – model, view, and controller – and therefore the application needed to be split in another way. The application was divided into several dependent plug-ins. Such division effectively enabled many people to work on the code simultaneously, did not cause any additional coding effort, and was invisible to the end-user. Model-View-Controller architecture integrates very well into the architecture of the Eclipse platform and therefore I would not call it a bad choice. Experiences with parallel development just further highlighted the need to keep the core small and implement additional features as dependent plug-ins. Carefully-planned access and extension points to the core plug-in (see section 2.3) proved to be extremely useful.

5.2 Achievement of objectives

The first objective was to develop a modular Eclipse plug-in for maintaining the Stylebase. The achievement of the objective was supposed to be measured by a tool demonstration, more precise criteria being defined as part of the requirement specification. In Autumn 2006, the tool was demonstrated at various events and superficially tested before a first beta release was published. In January 2007, students were used to test the tools both against the requirement specification and the end-user documentation. Even though some bugs were found, both testing and end-user feedback confirmed that the plug-in functioned as intended. First experience with parallel development supported the conclusion that plug-in was modular and easy to integrate with other plug-ins.

The second objective was to found a plug-in development project, which is listed at the Eclipse Plug-in Central (EPIC). Performance was to be measured by the number of downloads, as well as the amount of quality feedback received from users and contributors. The plug-in project was published at SourceForge at 19th of October 2006 and a few days after it was also listed at EPIC. By the time of writing, there has been approximately 450 downloads from different IP addresses. A quick look was taken at random SourceForge projects of roughly the same age and none of them had as many

downloads. Approximately 5 % of people who downloaded the product gave us feedback in some form. Most feedback was quite brief, only a couple of users got involved in longer discussions and a few reported bugs. There were no statistics available on average proportion of the number of contacts to the number of downloads. Generally speaking, it is expected that the vast majority of the users stay passive and only some become more active (Vainio & Vaden 2006). Nevertheless, new ways for encouraging communication could be explored in the future.

The third objective was to gain active users who contribute to the development of the plug-in. Performance was to be measured by the amount and quality of received contributions, if any. Completion of objectives during the time provided for this thesis work was not expected. Two students from Oulu University of Applied Sciences contributed to the development, the combined size of the contributions being 2–3 working weeks. Even though these contributions were initiated by myself, and did not genuinely rise from the needs of the community, they were useful in accumulating experience in working with external contributors. More contributions are expected to come in as the project matures. In fact, three new users have recently contacted me and expressed interest in participating in the development of the tool.

5.3 Future work

Feedback from user community was consistent on the point that the installation and configuration process of the plug-in needs to be simplified. It was decided that the Stylebase for Eclipse should offer its users a choice between a file system based version and a DBMS based version of the tool. Current version requires all users to install a MySQL database which makes the installation process too time-consuming and error-prone. Furthermore, there are no benefits from the use of DBMS when a user wants to manage a pattern repository locally.

To support the new functionality, the database schema shall be redesigned so that all information is stored in the XML format. This facilitates us to implement the file-system based version smoothly, without being doomed to maintain two entirely different versions of the tool. It also improves extensibility of the database. The tool currently stores unparsed XML documents in one large text field. As the role of XML is growing, it might be necessary to find a more efficient way to store XML in a relational database. This is a challenge because MySQL and other open source database do not have built-in XML features, like some closed-source database systems have already. One solution could be an approach which uses middle-ware for storing and retrieving XML data into/from MySQL database (Kurt et al. 2004). Further study on the subject is needed.

The design and implementation of this variation point – the choice between file-system based and RDBMS-based implementation – will be coordinated as an open source project. Further marketing activities are required to attract external contributors. The development of the variation point shall be used as an industrial example to test the Integrability and Extensibility Evaluation method (IEE) developed at VTT. IEE is a scenario-based evaluation method for assessing how integrability and extensibility requirements are met in the software architecture.

6. Conclusions

This thesis work was a case study on contributing to Eclipse. The primary research question was *"How to contribute to Eclipse by initiating a new plug-in project"*? The study covered both technical and organizational aspects of the matter, i.e. development of the plug-in and founding an open source community. The primary research question was divided into two sub questions, which are answered below. Most interesting findings were related to various aspects of FLOSS development.

The first sub question was "*How to develop a modular Eclipse plug-in?"*. Modularity should be implemented on two levels. (1) Firstly, the internal architecture of a plug-in must be modular. This can be achieved by designing the architecture in accordance with a tested and commonly-approved architecture pattern. The model-view controller pattern was selected because it supports maintainability, facilitates parallel development and complies with the architecture of the Eclipse platform. (2) Secondly, the plug-in must provide pre-defined interfaces for the downstream plug-ins to use. This can be achieved by (a) building *access points* by implementing a well-defined programming interface and (b) defining *extension points* with the Eclipse extension point mechanism. Experiences highlighted that breaking the application into several dependent plug-ins is by far the most effective way to support parallel development. It was also noticed that code re-use does not necessary quicken the development process – its primary benefits are in learning and knowledge re-use.

The second sub question was *"How to launch a successful open source community?"* Launching an open source community is much more than just publishing source code on the Internet. In order to found an open source community one needs to (1) choose a license and apply it to the code artifact (2) build technical infrastructure to support communications and distributed development (3) market the project intensely. Conclusions based on experiences in community building are as follows.

GPL was selected as a license because it is well-known and encourages contributing extensions and improvements back to the community. When integrating open source components, resolving legal compatibility issues takes time and therefore should be taken into account in schedules. It is best to avoid subcomponents that have been published under non-standard licenses and, regarding the GPL license, one should only use open source components with GPL compatible licenses accepted by the Free Software Foundation. If closed-source forks or extensions are not seen as harmful to the community, a more permissive license could be chosen to diminish incompatibility issues.

For building the technical infrastructure, subscribing to a canned hosting facility is a good option – and the only option if resources are scarce. If a hosting facility reviews the source code of proposed projects, one should prepare for a lengthy acceptance process. It may become necessary to modify the source code as requested by the administrators of the respective hosting facility. The most popular hosting facility, SourceForge.net, does not do code review and therefore new projects get accepted quickly.

General purpose hosting facilities cannot be relied on as marketing channels. Marketing via other channels – such as OS related internet forums and seminars – deemed to be crucial for attracting users and contributors. The marketing activities caused an immediate tenfold increase in code downloads at the project website.

User community was responsive to the idea of maintaining an architectural knowledgebase. Essentially all negative feedback concerned the fact that the installation process was thought to be time-consuming. It seems that giving a good initial impression to users includes getting them started with only a few mouse clicks. The future development challenge is to ease the installation and configuration process of the Stylebase for Eclipse plug-in. As for research interests, the next goal is to use the plug-in as an industrial case to test an evaluation method for assessing extensibility and integrability of software architecture.

References

Printed References

Bass, L., Clements, P. & Kazman, R. 2004. Software Architecture in Practice. Second Edition. Boston: Addison-Wessley.

Beck, K. & Gamma, E. 2004. Contributing to Eclipse. Dr. Dobb's Journal 29(9), 74–78.

Birsan, D. 2005. On Plug-ins and Extensible Architectures. Queue 3(2),40–46.

Bondi, A. 2000. Characteristics of scalability and their impact on performance. In the Proceedings of the 2000 Workshop on Software Performance (WOSP2000).Ottawa, Canada.

Booch, G., Brown, A., Iyengar, S., Rumbaugh, J. & Selic, B. 2004, s. 18, 19. An MDA Manifesto. Business Process Trends/MDA Journal 2004(05).

Brooks, F. 1995. The Mythical Man-Month and Other Essays on Software Engineering. Second Edition. Boston: Addison-Wessley.

Bushmann, F., Meunier, R., Rohnert, H., Sammerlad, P. & Stal, M. 1996. Pattern Oriented Software Architecture: A System of Patterns. Chichester: John Wiley & Sons Ltd.

Clayberg, E. & Rubel, D. 2006. Eclipse. Building Commercial Quality Plug-ins. Massachusetts: Pearson Education, Inc.

Dibona, C. 2005. OS and Proprietary Software Development. In: Cris Dibona et al. (eds). Open Sources 2.0. Sebastopol: O'Reilly.

Fogel, A. 2005. Producing Open Source Software. How to run a successful free software project. Sebastopol: O'Reilly.

Frankel, D. 2003. Model-Driven Architecture, applying MDA to enterprise computing. Indianapolis: Wiley Publishing Inc.

Gallardo, D., Burnette, E. & McGovern, R. 2003. Ecipse in Action. A Guide for Java Developers. Greenwitch: Meaning.

Gamma, E. & Beck, K. 2004. Contributing to Eclipse – Principles, Patterns, and Plug-Ins. New York: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1994. Desing Patterns. Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Gebert, I. 2003. Open-Source Database Management Systems in Small and Medium-Sized Companies. Seminar Paper. Rostock: University of Rostock.

Gershenson, J., Prasad, G. & Zhang, Y. 2003. Product Modularity: definitions and benefits. Journal of Engineering Design 14(3), 295–313.

Goldman, R. & Gabriel, R. 2005. Innovation Happens Elsewhere. Open Source as Business Strategy. San Fransisco: Elsevier.

Griffin, C. 2004. Tranformations in Eclipse. 18th European Conference on Object-Oriented Programming. Oslo, Norway.

Gruber, O. 2005. The Eclipse 3.0 platform: adopting OSGi technology. IBM Systems Journal 44(2), 289–99.

Hakala, A. 2005. Tool Integration in Eclipse. Jyväskylä: University of Jyväskylä. Department of Mathematical Information Technology. Pro gradu thesis.

Halpin, T. 2001. Information Modelling and Relational Databases. San Fransisco: Morgan Kauffman Publishers.

Helander, N. & Mäntymäki, M. 2006. Empirical Insights on Open Source Software Business. BRC Research Report #34. Tampere: Tampere University of Technology and the University of Tampere.

Henttonen, K. & Matinlassi, M. 2007. Contributing to Eclipse: A Case Study. In the Proceedings of the 2007 Conference on Software Engineering (SE2007). Hamburg, Germany.

Hissam, S. & Weinstock, C. 2005. Making Lighting Strike Twice. In: Feller, J. et al. (eds.) Perspectives on Free and Open Source Software. Massachusetts: Massachusetts Institute of Technology, MIT Press Ltd., 143–159.

IEEE Standards Committee 2000. IEEE Recommended Practice for Architecture Description of Software Intensive System (IEEE1471-2000). New York: IEEE Computer Society.

Kidane, Y. & Gloor, P. 2005. Correlating Temporal Communication Patterns of the Eclipse Open Source Community with Performance and Creativity. In: Proceedings of NAACSOS Conference, Notre Dame. North American Association for Computational Social and Organizational Science.

Kurt, A., Sevkli, Z. & Mercan, M. 2004. A middleware approach to storing and querying XML documents in relational databases. Advances in Information Systems: Lecture Notes in Computer Science 3261(1), 223–233

Matinlassi, M. 2005. Quality-Driven Software Architecture Model Transformation. In Proceedins of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05).

Matinlassi, M. 2006. Quality-Driven Software Architecture Model Tranformation. Towards Automation. Espoo: VTT Publications.

Matinlassi, M. 2007. Role of Software Architecture in Open Source Communities. Accepted to the proceedings of Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007). Mumbai, India.

Matinlassi, M. & Henttonen, K. 2007. Submission for COSI project deliverable D.3.3.7. Case Study Selection.

Matinlassi, M. & Kalaoja, J. 2002. Requirements for Software Architecture Modelling. In Proceedings of the Workshop of Software Modeling Engineering of UML 2002. Dresden, Germany.

Matinlassi, M., Niemelä, E. & Dobrica, L. 2002. Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. Espoo: VTT Technical Research Centre of Finland, VTT Publications.

Matinlassi, M. & Niemelä, E. 2003. Impact of Maintainability on Component based Software Systems. In: Proceedings of 29th EUROMICRO Conference (EUROMICRO'03). Turkey.

Merilinna, J. 2005. A Tool for Quality-Driven Architecture Model Transformation. Espoo: VTT Publications 561. 106 p. + app. 7 p.

Miller, J. & Mukerji, J. 2003. MDA Guide Version 1.0.1. Massachusetts: Object Management Group.

MySQL Ab 2005. Inside MySQL 5.0. A DBA's Perspective. MySQL Business White Papers 2005(10). Ordered at http://www.mysql.com/why-mysql/white-papers/mysql_wp_inside50.php.

Niemelä, E. & Immonen, A. 2007. Capturing the Quality Requirements of Product Family Architecture. Accepted to the Journal of Information and Software Technology. Elsivier.

Niemelä, E., Kalaoja, J. & Lago, P. 2003. Toward an Architectural Knowledgebase for Wireless Service Engineering. IEEE Transactions of Software Engineering.

Niemelä, E., Matinlassi, M. & Immonen, A. 2004. Quality-driven development of software product family. Espoo: VTT Publications . Available at http://virtual.vtt.fi/qada/images/qada_esite_final.pdf.

O'Riordan, C. 2006. Speech by Eben Moglen (Section 7e). The Transcript of the the Opening Session of the First International GPLv3 Conference on January 16th 2006. Dublin: Irish Free Software Organization. Available at http://www.ifso.ie/documents/gplv3-launch-2006-01-16.html.

Raymond, E. 2001. The Cathedral and the Bazaar. Sepastobol: O'Reilly. Partially available at at http://catb.org/~esr/writings/cathedral-bazaar/.

Robbins, J. 2005. OSSE Practices by Adopting OSSE Tools. In Compilation Joseph Feller (edit) Perspectives on Free and Open Source Software. Massachusetts: Massachusetts Institute of Technology, MIT Press, 245–254.

Rowan, W. 2006. Open Source Development – An Introduction to Ownership and Licensing Issues. Oxford: University of Oxford. Available at http://www.oss-watch.ac.uk/resources/iprguide.xml.

Scacci, W 2002. Understanding the requirements for developing open source software systems. IEEE Software.

Selic, B. 2003a. Model-Driven Development of Real-Time Software Using OMG Standards. In the Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03). Washington: IEEE Computer Society.

Selic, B 2003b. The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25.

So, H. 2005. Construction of an Evaluation Model for Free/Open Source Project Hosting Sites. Melbourne: RMIT University.
Available at www.ibiblio.org/fosphost/eval_fosphost090.pdf.

Souza, B. 2005. How Much Freedom Do You Want? In Compilation Chris Dibona et. al.(edit.) Open Sources 2.0. Sebastopol: O'Reilly, 219–224

Vainio, N. & Vaden, T. 2006. Sociology of Free and Open Source Software Communities: Motivations and Structures. In compilation Niina Helender & Hanna Martin-Vanhanen (eds.) Multidisciplinary Views to Open Source Software Business. BRC Research Report #33. Tampere: Tampere University of Technology and the University of Tampere, 10–19.

Varhol, P. 2006. Leaving the Crystall Ball Behind in App Development. Eclipse Review 1(3), 29–32.

Electronic References

Borland Software Corporation 2006. Borland® Together® technologies. http://www.borland.com/us/products/together/index.html#eclipse. Date of data acquisition 1 January 2007.

Deitel 2007. Open Source Resource Center. Open source conferences. http://www.deitel.com/OpenSource/OpenSource_ResourceCenter_Page10.html#Conferences. Date of data acquisition 16 March 2007.

Eclipse Foundation 2006a. About Us. http://www.eclipse.org/org/ Date of data acquisition 1 November 2006.

Eclipse Foundation 2006b. Eclipse Membership Applicaton of Change in Reprentation.http://www.eclipse.org/membership/become_a_member/Membership%20Application.pdf. Date of data acquisition 26 October 2006.

Eclipse Foundation 2006c. Eclipse Rights by Membership Catogory. http://www.eclipse.org/membership/become_a_member/How2Join%20Eclipse%20Rights%20by%20Membership%20Category.pdf. Date of data acquisition 25 October 2006.

Fleury, M. & Lindfors, J. 2001. Enabling component architectures in JVMX. http://www.onjava.com/pub/a/onjava/2001/02/01/jmx.html. Date of data acquisition 30 June 2006.

Free Software Foundation 2006a. Free Software Definition. http://www.gnu.org/philosophy/free-sw.html Date of data acquisition 11 November 2006.

Free Software Foundation 2006b. Various licenses and comments about them. http://www.fsf.org/licensing/licenses/index_html Date of data acquisition 20 October 2006.

Free Software Foundation 2006c. Savannah Services and Requirements. https://savannah.nongnu.org/register/requirements.php Date of data acquisition 20 October 2006.

Free Software Foundation 2006d. How to use GPL and LGPL. http://www.fsf.org/licensing/licenses/gpl-howto.html Date of data acquisition 27 October 2006.

Gabriel, R. & Joy, W. Sun Community Source Licensing (SCSL) – Principles. http://www.sun.com/software/communitysource/principles.xml. Date of data acquisition 26 October 2006.

Heizman, D. 2003. An introduction to open computing, open standards, and open source. http://www-128.ibm.com/developerworks/rational/library/1303.html. Date of data acquisition 16 March 2007.

Henttonen, K. Incompatibility with GPL. Newsgroup jdom- interest@jdom.org. 28 September 2006. Available at http://www.jdom.org/pipermail/jdom-interest/2006-September/015549.html

Henttonen, K. Stylebase for Eclipse. Free software for maintaining an architectural knowledge base. http://stylebase.sourceforge.net. Date of data acquisition 15 December 2006.

IBM Corporation and others 2003. Eclipse Plug-in Developer Guide. Eclipse Platform Help System. Available also at www.eclipsehelp.org.

ITEA 2006. COSI project. http://itea-cosi.org Date of acquisition 27 October 2006. Object Management Group 2004. MDA Success Story. Model Driven Software Development and Offshore Outsourcing.
http://www.omg.org/mda/mda_files/M1Global.htm. Date of data acquisition 24 August 2006.

Open Source Initiative. 2006. Open Source Definition.
http://www.opensource.org/docs/definition.php. Date of data acquisition 28 June 2006.

Open Source Watch. 2006. What is open source software? http://www.oss-watch.ac.uk/resources/opensourcesoftware.xml. Date of data acquisition 27 October 2006.

Raymond, E. & Moen, R. 2006. How to Ask Smart Questions The Smart Way. http://catb.org/esr/faqs/smart-questions.html. Date of data acquisition 27 October 2006.

Robson, S. [task #5865] Submission of Stylebase. Recipients: Katja Henttonen and savannah-register-public@gnu.org. 24 September 2006. Available at http://www.mail-archive.com/savannah-register-public@gnu.org/msg06275.html

Robson, S. [task #5865] Submission of Stylebase. Recipients: Katja Henttonen and savannah-register-public@gnu.org. 12 October 2006. Available at http://www.mail-archive.com/savannah-register-public@gnu.org/ msg06347.html

Robson, S. [task #5865] Submission of Stylebase. Recipients: Katja Henttonen and savannah-register-public@gnu.org. 12 December 2006. Available at http://www.mail-archive.com/savannah-register-public@gnu.org/msg06777.html

Sametinger, J. Re: Can we continue the development of PSE?. Recipient: Katja Henttonen 15 May 2006.

Smith, D. RE: Question on Eclipse membership. Recipient: Katja Henttonen. 2 October 2006.

Interviews

Niemelä, Eila, Research Professor, VTT Technical Research Centre of Finland. Interview on 15 May 2006.

Matinlassi, Mari, Project Manager, VTT Technical Research Centre of Finland. Interviews on 18 May 2006 and 4 September 2006.

Merilinna, Janne, Research Scientist. VTT Technical Research Centre of Finland. Interview on 20 May 2006 and 3 August 2006.

Tarvainen, Pentti, Serior Research Scientist. VTT Technical Research Centre of Finland. Interview on 20 September 2006.

Appendix 1: Evaluation of existing pattern management plug-ins

The following table lists the main characteristics of the existing, open source Eclipse plug-ins that include pattern management features.

Table 1. Summery of the pattern tools evaluated.

IBM Design Pattern Toolkit	**License:** CPL 1.0
Features	The tool generates applications based on customizable model-driven architecture patterns. The application model is implemented as an XML file and then used for code generation and validation. Tool support to pattern distribution is included.
Comments	All generated applications have a Model-View-Controller structure. Models are defined in a rather complex and non-standard manner, which is not compatible with any UML tool. There is an OSS community, mostly consisting of IBM developers.
Pattern Box	License: none
Features	Pattern Box generates source code from predefined, well-known design patterns (e.g. Singleton, Iterarator, Observer). Users cannot create new pattern descriptions. They neither have much control on the usage of existing patterns.
Comments	There is no OSS community. However, users are encouraged to sent new pattern templates to project maintainers by e-mail. The plug-in does practically nothing but provides code templates.
J2EE Design Pattern Generator	**License:** CPL 1.0
Features	The tool generates the code of J2EE design patterns and includes support for J2EE refactorings. It has a user friendly wizard for creating new pattern descriptions. Pattern descriptions are stored in XML files.
Comments	Pattern descriptions are non-standard and not compatible with any UML tool.There is only one developer in the associated OSS community.The tool is limited to J2EE
PatternBox2	**License:** GPL 1.0
Features	BatternBox2 generates source code from design motifs described in PADL (Pattern and Abstract-level Description Language). There are two other OSS-tools available from the same author: 1. PITJ which analyzes source code by locating pattern's used. 2. Caffeine which analyzes the execution of a Java program.
Comments	Description language is non- standard and extremely complex. It is a meta-model devised for author's Ph.D. thesis There is no OSS community and none of the tools are actively maintained by the author. It seems that the tool was created to demonstrate subjects of a thesis rather than to serve ordinary users.
Pattern Support for Eclipse (PSE)	License: none
Features	PSE generates source code from pattern descriptions stored in a XML file. The generator prompts user to assign component roles and their relationships (allowing only choices compatible with the pattern description). The tool provides view for navigating the defined design patterns and browsing their documentation. It also finds patterns from source code based on comments added by the generator.

Comments	The description language is non-standard and not compatible with any UML modeling tool. However, it seems efficient, simple and easy to learn. The tool which browses documentation is clearly similar to one needed by Stylebase. It seems that there was a lot of interest in the product and an active user community until authors abandoned the project. The tool does not run in the latest version of Eclipse.
SEDS Design Pattern Plug-in	**License:** GPL 1.0
Features	The website states: “Makes easier writing Java application with usage of design patterns. There are basic built in patterns … but also user can configure other patterns or some class hierarchy.”
Comments	The tool was not tested because the user interface is only available in Polish. However, the website promises that the tool will be internationalized soon.
Web Of Patterns (WOP)	**License:** CPL 1.0
Features	WOP scans pattern instances from Mandarax projects. The user can aggregate the results. It has a user-friendly wizard for creating pattern descriptions and associated application rules by using Mandarax classes. The website claims that the tool “has a pattern browser that can be used to browse online pattern repositories”
Comments	Mandarax in an open source tool for for defining, managing and querying a knowledge base of deduction rules. The pattern browser mentioned on the website does not seem to work. The product is related to a research project on software ontology. It does not have an open source developer community.

Appendix 2: Requirement specification

This document defines the functional and directive quality requirements for the base code of Stylebase for Eclipse. The sub-requirements which should be implemented and moderately tested before the project is published are marked with an asterisk (*). The requirements which were implemented by external contributors are marked with a hash (#).

Table 1. Functional and quality requirements.

Requirements	**Sub-requirements**	**Related quality-requirements**
Store architecture and design patterns	Store the following data for each pattern: name, description, type, data model, picture and guide (*)	**Extensibility**: Support to store macro architectures, micro architectures and reference architectures (and other, so far undefined, types of patterns) **Extensibility**: Support to store data model and guide in any selected form (Defaults will be XML and HTML respectively.)
	Manage two levels of repositories: local (and global/company (*)	
Provide a database interface for accessing the pattern repository	Implement interface functions such as select, update, insert, delete (*)	**Modularity/Maintainability**: Keep the database interface separate from other components e.g. user interface **Scalability**: Support for several different tools and several users per tool to access the knowledge base at the same time **Security**: User authentication, read/write permissions **Usability**: Simultaneous updates must be handled without errors (locking)
	Provide an initial pattern library containing popular patterns (*)	
Provide a graphical user interface (GUI) for browsing and updating the patter repository.	Provide a dialog for searching patterns (by name, description and quality attributes) and show search results in a view (*)	**Usability**: Browsing shall be easy and fast for end user satisfaction
	Provide a possibility to delete a pattern or update its properties (*)	**Security**: Permissions required
	Provide dialogs for downloading/uploading patterns to/from local file tree (*)	**Security/Usability**: Locking to prevent users from overwriting each others changes
	Provide a tool for browsing pattern's documentation (*)	**Usability**: Documentation must be accessible directly from the remote database (without exporting the file).

	Provide self-selecting help for using Stylebase for Eclipse (#)	**Integrability**: The help files shall extent the document tree of the Eclipse platform help **Usability**: Help files shall be accessible without network connection.
	Provide a dialog for adding and editing the definitions of quality attributes (#)	
Provide a pattern creation wizard	Control input to make sure that a pattern is inserted correctly	
	Provide a template for pattern's documentation	
	Save new pattern locally and, if requested, upload it into global/company level database	
Provide a GUI for configuring the plug-in (#)	Set configuration parameters such as database connection parameters, default file types and file locations (#)	**Integrability**: GUI shall be an extension into the preferences page of the Eclipse platform

Appendix 3: Database schema

This document contains a detailed description of the schema of the database which is used by Stylebase for Eclipse. Figure 1 presents the relational model of the database. Fields which form a primary key are underlined. Tables 1–3 list columns of each table and describe the information they contain.

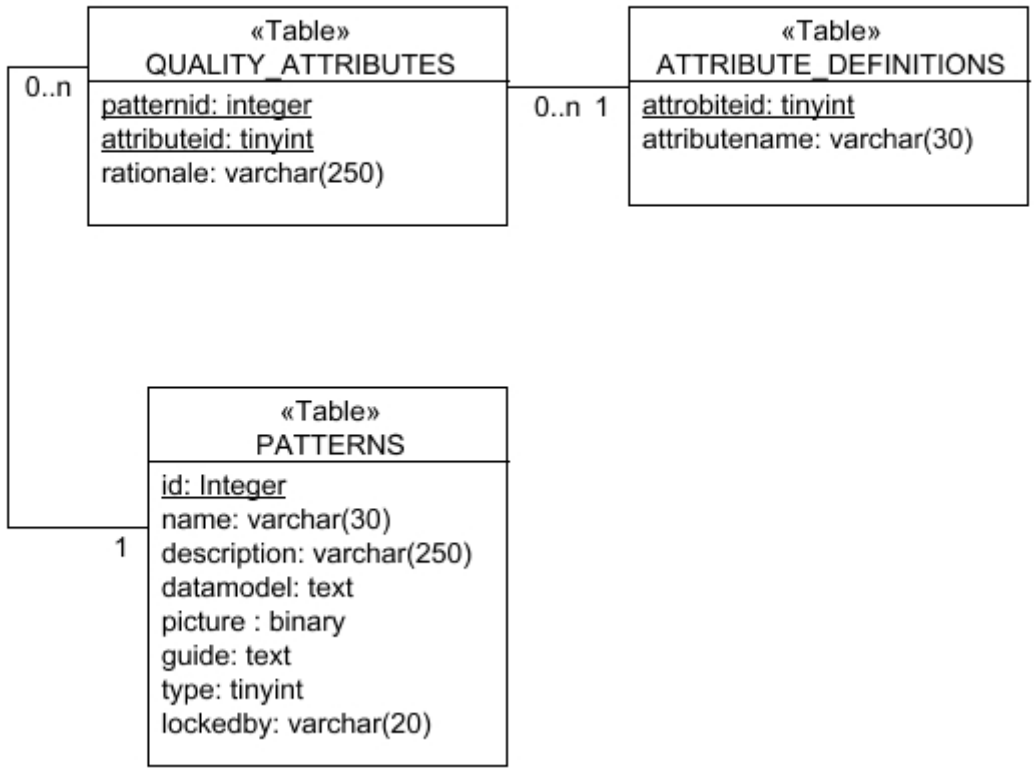

Figure 1. Database schema of Stylebase plug-in.

Table 1. Descriptions of fields in table "Patterns".

TABLE PATTERNS		
Column name	Value range	Purpose
id	integer from 0 to 4294967295	Contains a unique id which is used to identify the pattern. When a new pattern is inserted, the field is automatically assigned to the next available value.
name	up to 30 characters	Contains the primary name by which the pattern is known. Values in the field "name" and "type" must form a unique combination.
description	up to 250 characters	May be used to describe the intent of the pattern and/or list its alternative names. Users may search patterns based on any word on this field. If preferred, may only contain a list of search words.
guide	ascii data up to 2^{16} bytes	Stores natural language documentation for the pattern. The documentation should include, for example, component descriptions, motivation, applications and instructions on usage.
datamodel	ascii data up to 2^{16} bytes	Stores pattern's data model in XML or other mark-up language used by modeling tools. It is typically an UML diagram saved in XMI format.
picture	binary data up to 2^{16} bytes	Stores pattern's data model as a picture, typically in GIF or JPG format. The picture is meant to complement documentation and it's not used by the application logic.
type	integer from 0 to 255	Integer representing pattern type. Values currently in use: 1=design pattern 2=architecture pattern
lockedby	up to 20 characters	Stores user name of the user who is currently holding update lock on the pattern. Empty string indicates no locking.

Table 2. Descriptions of fields in table "Attribute_definitions".

TABLE ATTRIBUTE_DEFINITIONS		
Column name	Value range	Purpose
attributeid	integer from 0 to 255	Contains a unique id which is used to identify the quality attribute. When a new attribute definition is inserted, the field is automatically assigned to the next available value.
name	up to 30 characters	Contains the name of the quality attribute (e.g. "portability" or "reliability")

Table 3. Descriptions of fields in table "Attribute_definitions".

TABLE QUALITY_ATTRIBUTES		
Column name	Value range	Purpose
attributeid	(see above)	Reference to column "attributeid" in table "attribute_definitions".
patternid	(see above)	Reference to column "id" in table "patterns".
rationale	up to 250 characters	Brief explanation on why the pattern has been associated with the quality attribute.

Appendix 4: Concrete architecture description

This document describes the static architecture of the Stylebase for Eclipse on concrete level. It should be read along with the conceptual level architecture description in chapter 3.4. The document presents the static structure of Java classes and explains their responsibilities and interrelations.

In the Stylebase for Eclipse source code, Java classes are arranged into packages called "model", "view" and "controller", corresponding to conceptual level components. The below diagram illustrates the packages and names of concrete interface objects between them. Responsibilities of the interface objects are discussed in subsections that describe the concrete structure of each package.

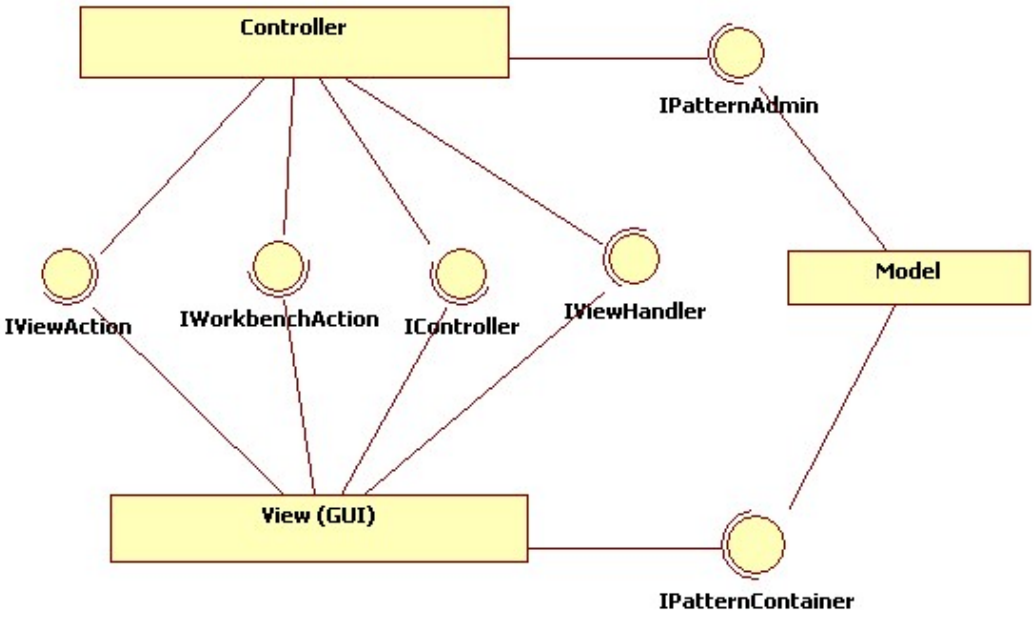

Figure 1. Packages and Interfaces of the Stylebase plug-in.

View

View component provides the graphical user interface, which consists of views and dialog windows. Every view with actions is dependent on its controller interface. Every dialog is dependent on an action, to which it sends user input events for processing via interfaces IWorkBenchAction and IViewAction Events which cause nothing but a cosmetic change on display (e.g. enable a menu option) are not sent to the controller, but handled inside the view component. Instead of a direct association, dialogs communicate with their parent view via an interface (IViewHandler). This makes it

possible to open the same dialog from more than one view. Views and dialogs receive their content from the model component via IPatternContainer interface.

The below diagram illustrates the structure of the view component.

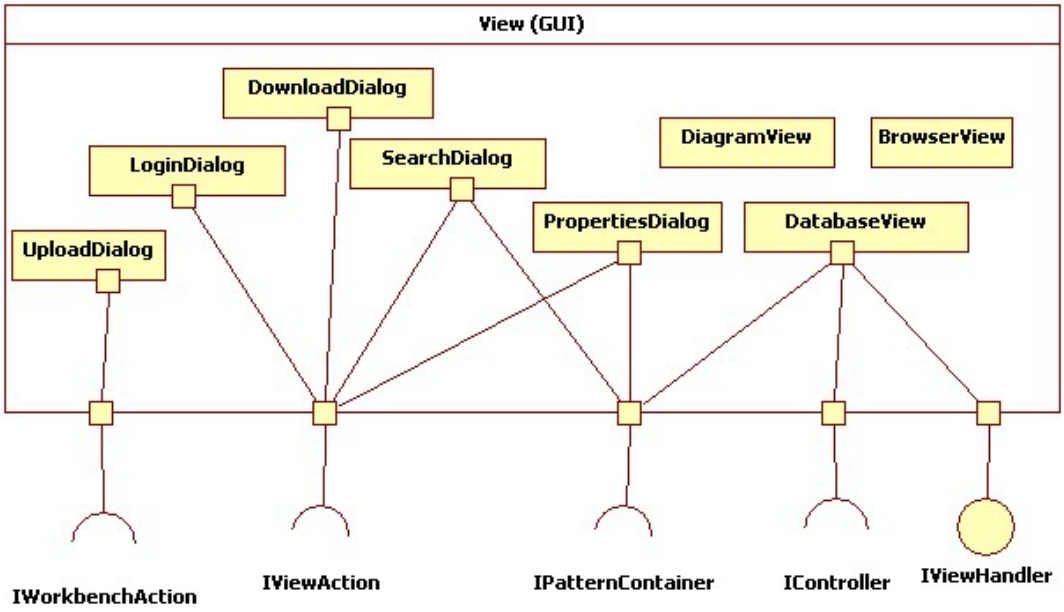

Figure 2. Structure of the view component.

The below diagram presents the detailed structure of the classes, including the real names of methods and attributes.

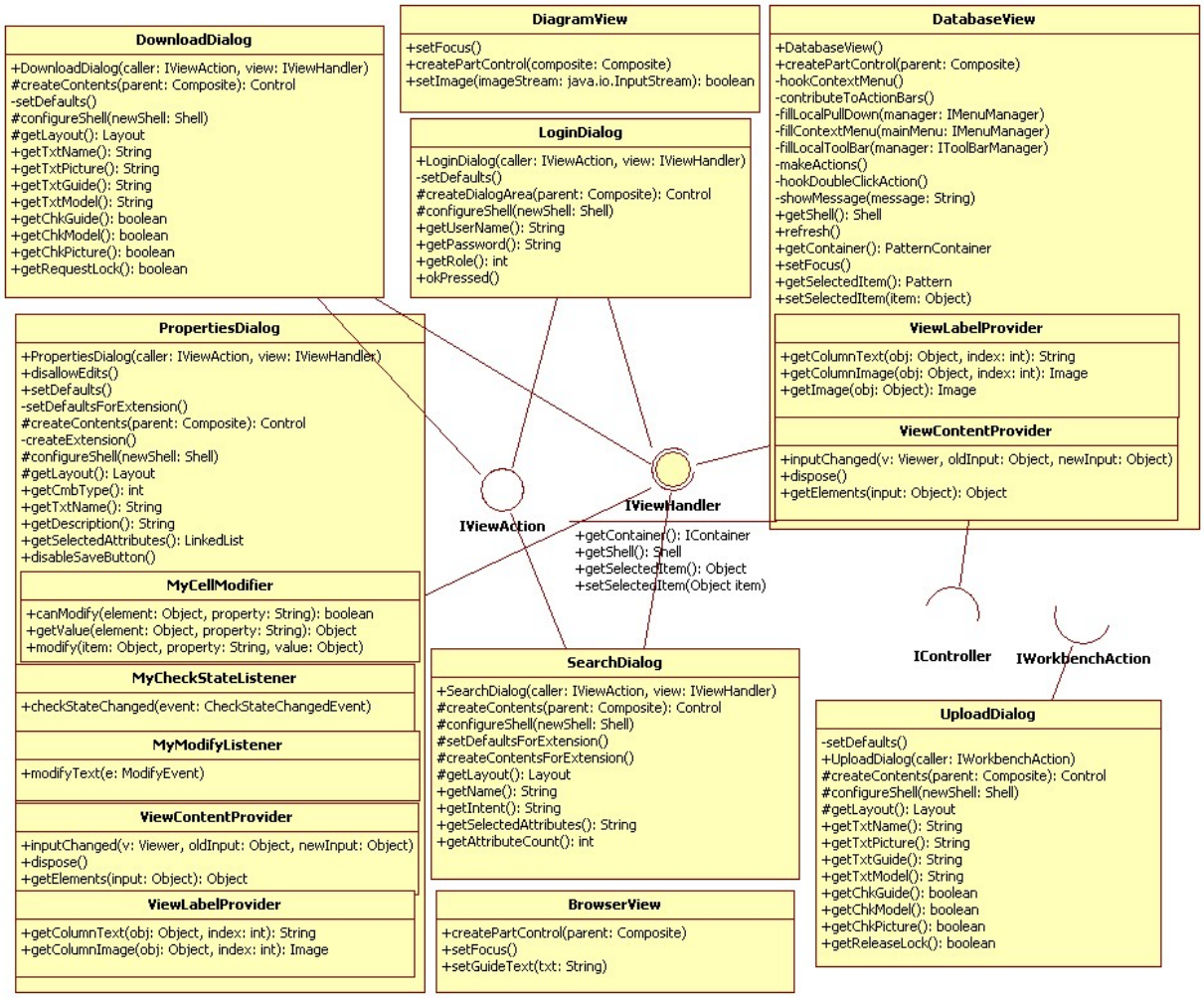

Figure 3. Class diagram of the View component.

Controller

Controller component manages actions. In Eclipse IDE, an action is a non-user interface part of the command that can be run by a user, usually associated to a GUI element like a toolbar button or menu. Actions are implemented as Java classes with certain compulsory methods (e.g. run). In the implementation of Stylebase for Eclipse, the action classes are reusable: it is possible to associate the same action with more than one view.

There is one controller class per each view with actions. Because the base code contains only one such view, there is only one controller classs: DatabaseViewController. This class contains all actions which can be launched from the main view. The actions are created and destroyed at the same time with its controller object.The upload action is not managed by the controller, because it is launched from the Eclipse work bench. It is

declared as a workbench extension and its constructor is called implicitly by Eclipse. IActions modify the model component via IPatternAdmin interface.

Figure 4 illustrates the structure of the controller component.

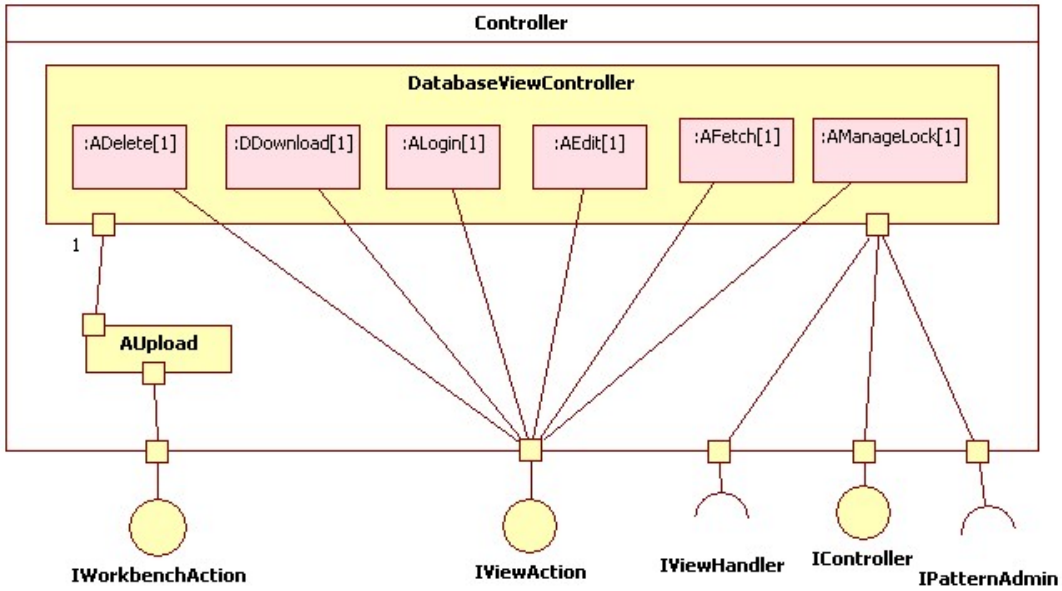

Figure 4. Controller composite structure.

The below diagram presents the detailed structure of the classes, including the real names of methods and attributes.

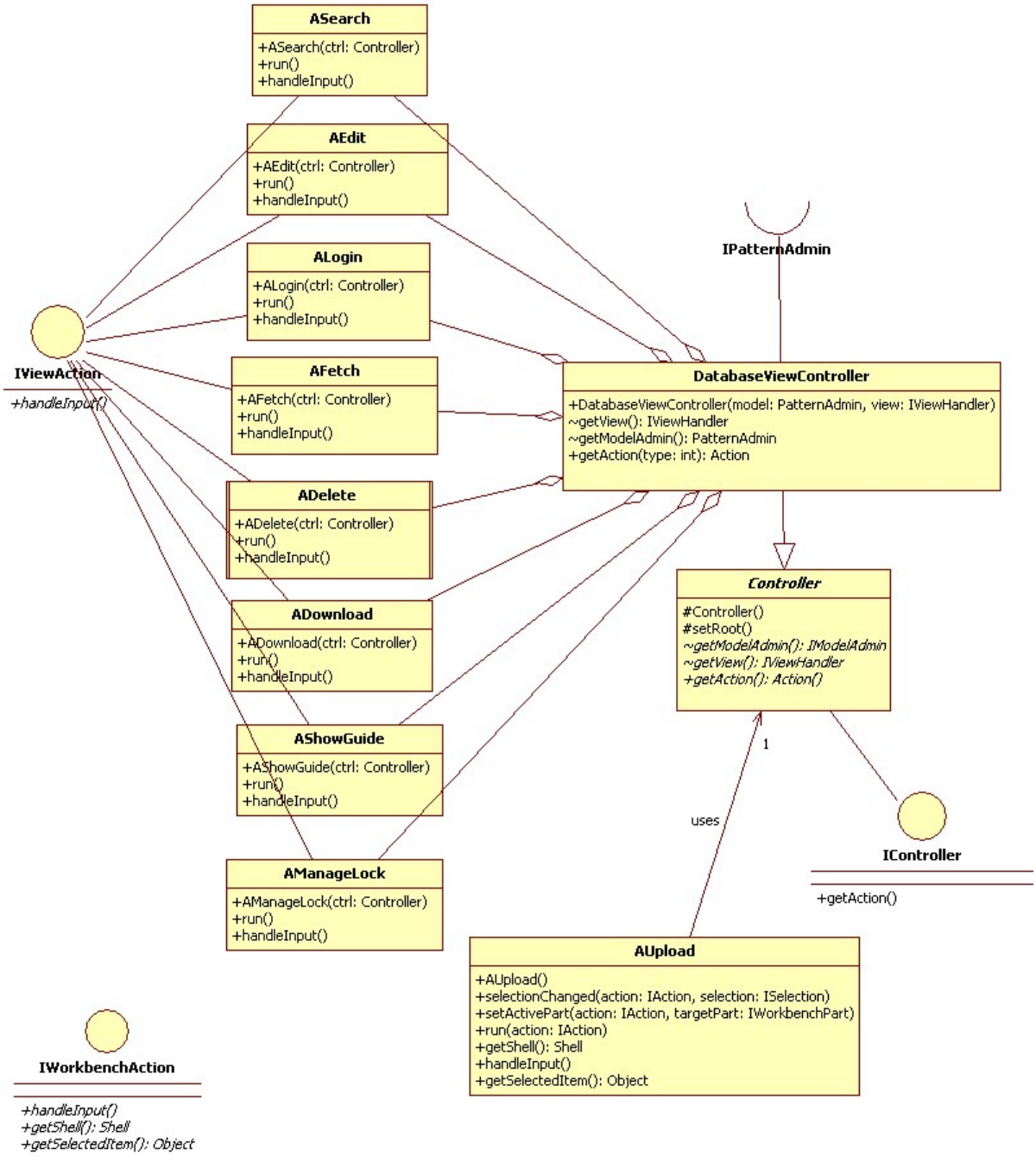

Figure 5. Class diagram of the Controller component.

Model

The model component is divided into two singleton classes, this is classes that can have only one instance during program's execution. The methods which manipulate the data are implemented in a subcomponent called "PatternAdmin". It connects to a remote database and retrieves, inserts and updates the data according to the instructions

received from the controller. “PatternContainer” is a passive storage object which keeps the pattern data in memory and provides contents for the view. The instance of the admin class is the only object which is allowed to change the data in the container. When the plug-in is started, the admin class retrieves information from the database and fills the container. Later on, it updates the database and the container according to the instructions received from the controller component. Figure 6 shows the structure of the model component.

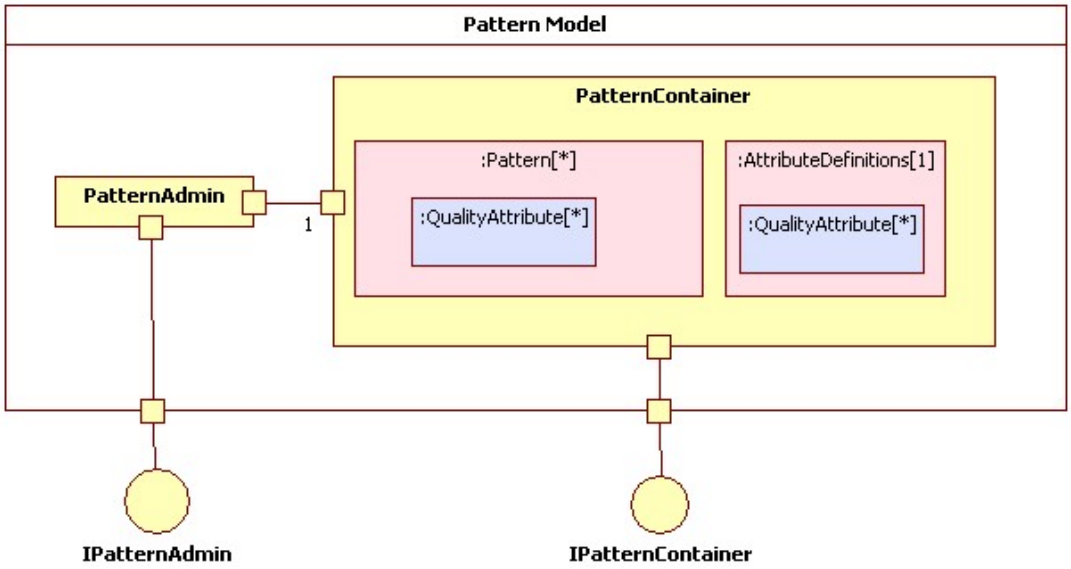

Figure 6. Model composite structure.

The below diagram presents the detailed structure of the classes, including the real names of methods and attributes.

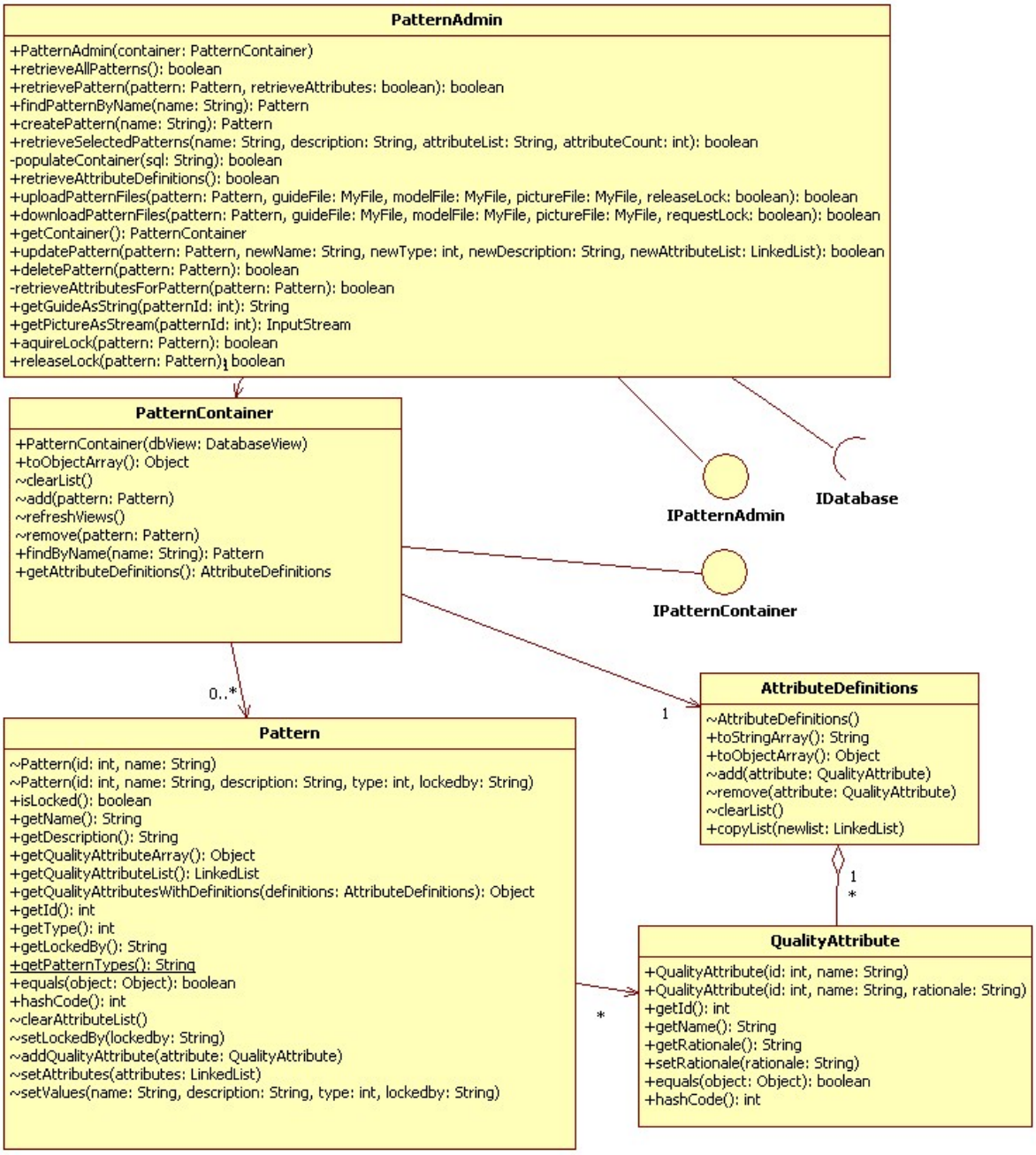

Figure 7. Class diagram of the model component.

System components

In addition to the model, view and controller, the plug-in implementation contains two other components. The database component encapsulates database access functions. Other components use it via an interface. System component is a package of classes that implement system methods. These include, e.g., methods for writing to log file, reading configuration file and managing session parameters. Each class has only one instance and methods can be called from any part of the system.

The below diagram presents the detailed structure of the classes, including the real names of methods and attributes.

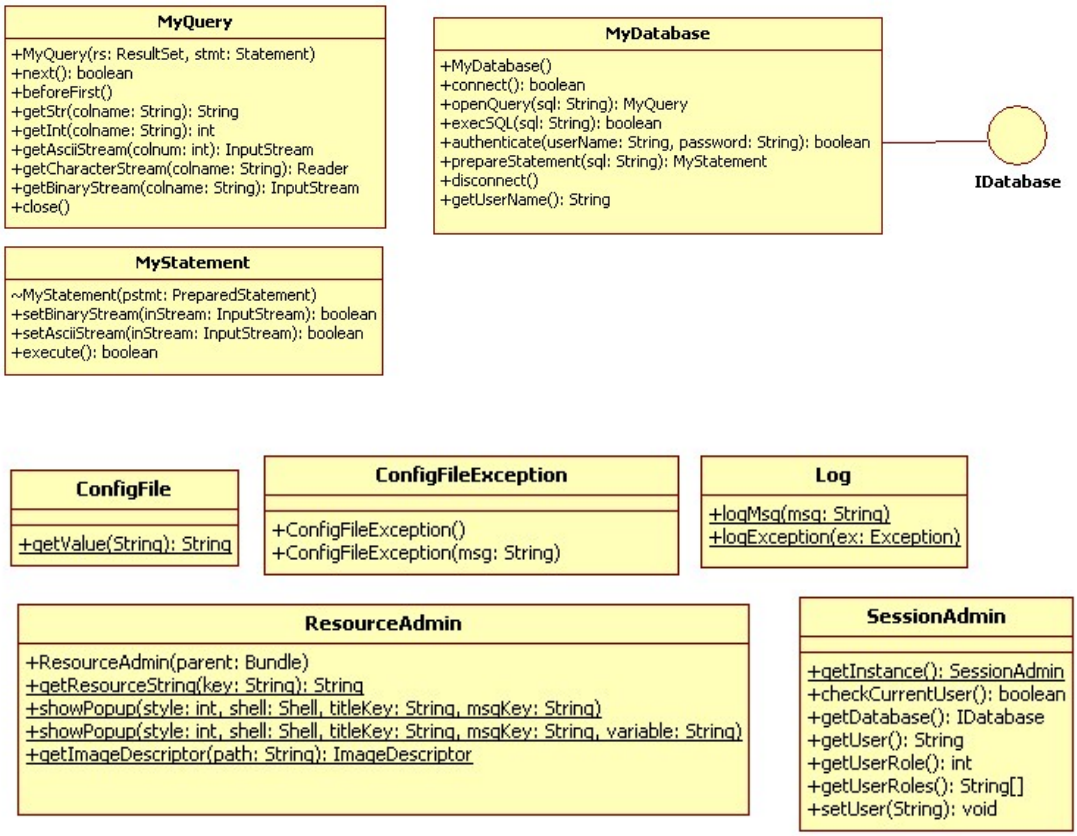

Figure 8. Class diagram of the system component.

Access points

In addition to serving as "gateways" of communication between the main components of the plug-in, interfaces server as access points to downstream plug-ins. The following table maps the interface classes into the access points defined at the conceptual level.

Table 1. Concrete interface objects mapped to access points.

Access point	Concrete Interface Object
Model	IPatternAdmin, IPatternContainer
Controller	IController, IViewAction, I WorkbenchAction
Database	IDatabase

Series title, number and
report code of publication

VTT Research Notes 2387
VTT-TIED-2387

Author(s)
Henttonen, Katja

Title

Stylebase for Eclipse

An open source tool to support the modeling of quality-driven software architecture

Abstract

Open source software has gained a lot of well-deserved attention during the last few years. Eclipse is one of the most successful open source communities providing an open development environment and an application lifecycle platform. Eclipse is a vendor-neutral platform for integrating tools and services. My thesis work is a case study on contributing to Eclipse. The contribution is a software architecture tool called "Stylebase for Eclipse" which is implemented as an extension a.k.a. *plug-in* to Eclipse.

Quality-driven architecture design is an approach to software architecture design which emphasizes the importance of qualities. Qualities are non-functional characteristics of a software system such as security or maintainability. Stylebase is a knowledge base of *software patterns* and *architectural styles*. It stores information that helps a software architect in selecting patterns that best support the desired quality goals. Stylebase for Eclipse is a tool for browsing and maintaining the stylebase. The purpose of the tool is to improve the quality of design and increase information sharing and re-use of architectural models in development teams.

In the case study, the plug-in is first developed and, after that, a new open source community is formed around the plug-in project. In order to comply with the open source development model, *modularity* is treated as the most important non-functional requirement. In community building phase, efforts are concentrated on marketing the new open source project and creating a good technical infrastructure for it.

The most interesting experiences gained during the study are related to various aspects of open source development. They are – among others – re-using code from other projects, licensing issues, tools to facilitate distributed development, and attracting new users and developers.

ISBN
978-951-38-6925-0 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN	Project number
VTT Tiedotteita – Research Notes 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)	12370

Date	Language	Pages
May 2007	English, Finnish abstr.	61 p. + app. 15 p.

Name of project	Commissioned by

Keywords	Publisher
Eclipse, open source, modeling, software architecture, quality-driven	VTT Technical Research Centre of Finland P.O.Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Tiedotteita 2387
VTT–TIED–2387

Tekijä(t)
Henttonen, Katja

Nimeke

Tyylikanta Eclipseen

Avoimen lähdekoodin työkalu tukemaan laatuohjatun ohjelmistoarkkitehtuurin mallintamista

Tiivistelmä

Avoimen lähdekoodin ohjelmistot ovat saanet paljon ansaittua huomiota viime vuosina. Eclipse on yksi menestyneimmistä avoimen lähdekoodin yhteisöistä, joka tarjoaa avoimen kehitysympäristön ja sovelluskehyksen. Eclipse on toimittajariippumaton integrointialusta työkaluilulle ja palveluille. Opinnäytetyöni on tapaustutkimus, jossa tehdään *kontribuutio* eli lahjoitetaan kehitystyötä Eclipse-yhteisölle. Kontribuutio on ”*Stylebase for Eclipse*” -niminen työkalu ohjelmistoarkkitehdeille, joka toteutetaan Eclipse-laajennoksena.

Laatuohjattu arkkitehtuurisuunnittelu on lähestymistapa, joka painottaa laatuominaisuuksien merkitystä ohjelmistoarkkitehtuurin suunnittelussa. Laatuominaisuuksilla tarkoitetaan ohjelmiston ei-toiminnallisia piirteitä, esimerkiksi tietoturvallisuus tai ylläpidettävyys. Stylebase eli *tyylikanta* on tietovarasto, joka sisältää arkkitehtuurityylejä ja suunnittelumalleja. Tyylikantaan tallennetun tiedon avulla ohjelmistoarkkitehti osaa valita ne tyylit ja mallit, jotka parhaimmin tukevat määriteltyjä laatutavoitteita. Stylebase for Eclipse on työkalu tyylikannan selaamiseen ja hallintaan. Työkalun tarkoitus on parantaa suunnittelutyön laatua sekä edistää tiedon vaihtoa ja arkkitehtuurimallien uudelleenkäyttöä kehitystiimeissä.

Tutkimuksessa laajennos ensin rakennetaan ja sitten perustetaan oma avoimen lähdekoodin yhteisö jatkamaan laajennoksen kehitystyötä. *Modulaarinen* rakenne on laajennoksen tärkein ei-toiminnallinen ominaisuus, koske se luo edellytykset avoimen lähdekoodin kehitystyölle. Yhteisön perustamisvaiheessa keskitytään projektin markkinointiin sekä sen tarvitseman infrastruktuurin rakentamiseen.

Mielenkiintoisimmat kokemukset ja tulokset liittyvät avoimen lähdekoodin kehityksen eri piirteisiin. Näitä ovat mm. ohjelmakoodin uudelleenkäyttö, lisensiointikysymykset, hajautettuun kehitystyön apuvälineet sekä uusien käyttäjien ja kehittäjien löytäminen.

ISBN
978-951-38-6925-0 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN	Projektinumero
VTT Tiedotteita – Research Notes 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)	12370

Julkaisuaika	Kieli	Sivuja
Toukokuu 2007	Englanti, suom. tiiv.	61 s. + liitt. 15 s.

Projektin nimi	Toimeksiantaja(t)

Avainsanat	Julkaisija
Eclipse, open source, modeling, software architecture, quality-driven	VTT PL 1000, 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374

VTT RESEARCH NOTES 2387 Stylebase for Eclipse. An open source tool to support the modeling of quality-driven software architecture

ISBN 978-951-38-6925-0 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0865 (URL: http://www.vtt.fi/publications/index.jsp)