Integrability and Extensibility Evaluation from Software Architectural Models – A Case Study

K. Henttonen*, M. Matinlassi, E. Niemelä and T. Kanstrén

VTT Technical Research Centre of Finland

Abstract: Software systems are composed of components acquired from different sources, e.g. subcontractors, component providers, and open source software providers. Therefore, integrability is one of the most important qualities in software development. Extensibility is especially important in open source software systems because they evolve according to the needs of the user community and often into a direction not originally foreseen. Integrability evaluation refers to testing if separately developed components work correctly together. Extensibility evaluation focuses on how new features, originated from customers' demands or new emerging technologies, could easily be developed and exploited in systems without losing existing capabilities. The impact of changes to the system also has to be estimated. This can be done by a method called IEE, which enables extensibility and integrability evaluation from software architectural models. The contribution of this paper is to introduce the IEE method and illustrate how it is to be used with a real world case study. In the case study, we applied the IEE in evaluating the architecture of an existing open source tool. Evaluation revealed a need to introduce two new extension points to the architecture and also that an integration framework is needed to integrate the tool under evaluation with other supporting tools.

Keywords: Integrability, extensibility, evaluation, quality, modeling, software architecture, quality-driven, software family, open source.

INTRODUCTION

Software systems, especially software families, are integrated systems based on proprietary components, commercial components, open source components and specific components of 3rd parties adapted to the needs of particular software systems [1,2,3]. In this context, software family refers to a collection of software-intensive systems that share common features and architectural concepts in order to fulfil a specific mission. The quality of the used heterogeneous components has a strong influence on the quality of target systems, because the scope of a software family architecture is broader than that of a single system architecture. The quality of a family architecture is of high importance because it is a long term investment; the life span of a family architecture ranges from 5 to 25 years. The aim of integration is to cut down development costs and to shorten the time to market by using components that can be integrated together for achieving the desired functionality. Because of changing customer needs, there is a challenge to keep software architecture stable and flexible at the same time. It is not easy to continuously provide new products to the markets based on emerging technologies, and still to remain competitive in terms of product quality. In software families, the trend is to concentrate on the differentiating parts of systems and to use the 3rd party software for commonalities [4]. Thus, integrability may be even more important in the development of software intensive systems in the future than it is today. Because of the unpredictability of changes in customer needs and markets, architecture has to remain flexible, allowing new features and components to be added to software systems during their evolution.

*Address correspondence to this author at the VTT Technical Research Centre of Finland; E-mail: katja.henttonen@vtt.fi

The importance of extensibility and integrability is particularly evident in the context of open source software. Open source software development is global movement that seems to be changing the way of software development as we have known it so far. Open source software is made available with its source code and under a license that allows anyone to use, modify, and distribute the modified or unmodified version of the software [5]. It is mostly developed by volunteers who work in a distributed environment [6]. Eric Raymond [7] resembled the open source development into "a great babbling bazaar of differing agendas and approaches". In other words, open source software systems are constructed from modules developed by business and community actors whose skills, interests and agendas may vary significantly. The requirements engineering process is typically incremental, user-driven and decentralized and therefore the requirements are most likely to increase beyond those foreseen [8,9].

Although there are a number of different quality evaluation methods and techniques available, e.g. for evaluating interoperability [10, 11], extensibility [12], architecture mismatches [11, 13] and multiple quality attributes [14], to our knowledge there is no method for integrability and extensibility evaluation that would cover software development from integrability and extensibility (IE) requirements specification to architecture design and that would enable quality evaluation from architectural models. Scenario development and scenario evaluation are the common activities for all scenario-based methods. The main differences between methods are; how early in the software architecture design the method is used, what quality attributes the method supports and how easy it is to apply and integrate to the design process [15].

Our contribution is an IEE (Extensibility and Integrability Evaluation) method that is an integrated part of the

1874-107X/07

QADA® (Quality-driven Architecture Design and quality Analysis) methodology [16]. If a software architecture has been developed in accordance with QADA, the use of the IEE method takes only some extra working hours. The method can be easily learned and adopted by architects and quality engineers, especially if already familiar with the principles of QADA. The method supports two quality attributes, integrability and extensibility, which are of great importance for evolvable software systems. QADA also provides other evaluation methods, e.g. RAP (Reliability and Availability Prediction) [17] and AEM (Adaptability Evaluation Method) [18], for other quality attributes. In this paper, the IEE method has been applied to a case called Stylebase for Eclipse[1]. The case study is an open source tooling environment for software architects and designers.

The structure of the paper is as follows. The background section introduces the topic by discussing the selected aspects of software architecture. After that an overview of the IEE method is provided, followed by the description of the case study. The case exemplifies how the method can be used to evaluate integrability and extensibility aspects from architectural models. The main sections in evaluation are impact analysis, quality and variability analysis, hierarchical domain analysis, scenario modeling and quality evaluation. Discussion summarizes our experiences on using the method and concludes the paper.

BACKGROUND

Software Architecture

A commonly agreed[2], short definition of software architecture is the structure of the software system including components and relationships. Further, in literature, there has been defined at least seven different *meanings* for software architecture. In general, architectural models *document* architecture to the body of knowledge for reusing the architecture at multiple levels of granularity [19, 20, 21]. Quite recently some guidelines for software architecture documentation, such as [20,22], have emerged.

Further, architecture models are *a manifestation of the earliest design decisions* [23,20,21] and *a means of abstraction* [20,24] to understand the system. Examples of design decisions are the decisions such as "we shall separate user interface from the rest of the application to make both user interface and application itself more easily modifiable". Manifestations of the design decisions are many and they may even be as small as definition of components and connectors.

Also, software architecture models can be seen as *the language for communication* [23,20,21,24]. The role of architecture models is also to *provide analysis opportunities at early stages of development* [19,20]. Architecture model is also an *expression of the system's evolution* [19,20] and *a management instrument* [20,24].

As the meanings of software architecture are many, the role of software architect has become very demanding. The level of abstraction has risen, required amount of cumulative knowledge has exploded and international and multicultural environments with geographically distributed development sites emphasize an ability to communicate ideas clearly. Clements *et al.* [26] made quite an extensive survey on the duties, skills and knowledge required from software architects today. The survey covered, e.g web pages, books, job descriptions and university courses on software architecture. This study considered that software architect and quality analyst play one role and therefore, the duties of software architect include project and requirements management and also architecture evaluation and analysis duties. In addition to communication skills, an architect needs the skill for abstraction, i.e. skills for handling the unknown and skills for handling the unexpected. These are two different but related skill sets. Skills are important but, useless without competent and appropriate knowledge on e.g. computer science, architecture concepts, technologies and platforms, programming and knowledge on organization's context and management.

Considering architecture concepts - among the most important ones are software patterns. Software patterns [23,26,27] encapsulate the idea of communicating insight and experience about common software engineering problems and their solutions [23]. Nowadays, software community is using patterns widely for software architecture and design. An architectural pattern expresses a fundamental structural organization schema for software systems, which consists of subsystems, their responsibilities and interrelations [26]. For example, layered architecture is a call-and-return style, when it defines an overall style to interact. When it is strictly described and commonly available, it is a pattern [26]. A design pattern is smaller in scale, describing a schema of communicating objects on design level. Design patterns are based on practical solutions implemented in mainstream programming languages [27]. Each software pattern implements tactics to achieve a particular goal (e.g. better performance or dynamic extensions) and also makes choices about tactics. Therefore, patterns are often concerned with different quality attributes and the design process involves making a choice of which patterns best provide the desired qualities [28].

Generally speaking, qualities, quality goals, quality attributes, quality requirements or, non-functional requirements, see e.g. [29,30,23,31], answer to the question *how well* whereas software functional requirements answer the question *what*. In the next section, it is discussed about quality-driven software architecture development i.e. an architecture design and analysis approach that is driven by quality goals.

Quality-Driven Software Architecture Development: Design and Analysis

Quality-driven software architecture development emphasizes the importance of qualities, wherein qualities refer to the non-functional properties of software products. The approach relies on gathering, categorizing and documenting quality properties as at least equally important requirements as functional requirements and constraints, and utilizing the gained knowledge in architectural design. The quality-driven design is further complemented with an architectural analysis. Architectural analysis is about testing the architecture model produced in the design, i.e. verifying whether the architecture meets the quality goals set in the very beginning.

[1]http://stylebase.sourceforge.net
[2]http://www.wikipedia.org

These two activities combined together form an interacting pair of activities in software architecture development[3].

The work described in this article is a part of a long-term research started in 2000 [32], namely the development of the QADA® (Quality Driven Architecture Design and Analysis) methodology. The development has been done in a sequence of various types of research projects involving several researches, each project and researcher focusing on certain part(s) of the methodology. The research approach of the whole concept is to create, validate and improve parts of the methodology as methods, techniques and realizations, to evaluate the parts and therefore to iteratively elaborate the methodology. Methodology parts are individual methods, wherein a method [33] denotes (1) an underlying model, (2) a language, (3) defined steps and ordering of these steps and (4) guidance for applying the method complemented with (5) tool support.

The focus of the QADA methodology is on identifying as many as possible of the design problems and quality goals in architecture design and analysis. In the design, this is achieved by identifying system stakeholders, analyzing target system quality goals from the point of view of several different stakeholders and describing the architecture and quality with models from various viewpoints so that the appropriate knowledge reaches each stakeholder. The analysis considers quality goals of architecture and products from the point of view of at least developers, users and customers.

Integrability and Extensibility as Quality Attributes

Integrability means an ability to make separately developed components of a system to work correctly together. Integrability is related to interoperability and again, interconnectivity. *Interoperability* is the ability of software to use the exchanged information and to provide something new originated from exchanged information whereas *interconnectivity* is the ability of software components to communicate and exchange information. Thus, interconnectivity is a prerequisite for interoperability and those two - interconnectivity and interoperability - are intertwined with functionality and visible at runtime [3].

Integrability has a decisive impact on the development and evolution of a system, due to which it should be taken into account as well as the other features of a system family, such as domain requirements, coarse grained architectural elements and the practices used for developing and maintaining a system family and deriving products from it. Interoperability is considered when components and their interactions are defined in detail and finally observed as executable models, simulations and running systems. *Extensibility* is the ability to extend a software system with new features and components without loss of functionality or qualities specified as requirements. In order to evaluate integrability and extensibility (IE), architecture characteristics should be identified from architectural models and components documented in a way that assist IE evaluation.

OVERVIEW OF THE IEE METHOD

This section provides an overview of the IEE method, one of the evaluation methods provided by the QADA methodology. The details will be clarified later when we explain how the method is applied in the case study. IEE is a scenario-based evaluation method. It is aligned with the principles of QADA and consists of the following three phases:

Phase 1: Defining quality goals and quality criteria.

Phase 2: Defining and modeling change scenarios for IE evaluation.

Phase 3: Evaluating integrability and extensibility of the family architecture from architectural models.

Fig. (**1**) presents the main activities of the IEE method defined by a UML2 activity diagram. The horizontal swim-lanes are named according to the engineering stakeholders or roles responsible for the defined activities and the vertical swim-lanes are named according to the main phases of the IEE method.

The first phase includes four activities: impact analysis, quality analysis, variability analysis and hierarchical domain analysis. Domain experts are responsible for impact analysis and quality analysis. In the impact analysis, the domain experts identify and elicit the interests of the business stakeholders and technical stakeholders, define the standards, regulations and practices to be followed in the domain. This activity results in a list of the stakeholders and quality goals. Software family architects identify and define variability of functional and non-functional capabilities inside a family (i.e. variability analysis), and categorize capabilities to service taxonomy taking into account the defined functional capabilities, quality goals, variability and commonality of the capabilities (i.e. hierarchical domain analysis). Phase 1 results in a list of prioritized quality criteria against which the architecture is evaluated. Instructions how to define quality goals and quality criteria, and how to represent the required and provided quality properties in architectural models are given in [34]. The article describes thoroughly the QRF (Quality Requirements of a software Family) method and produces an evidence how it is applied in the context of a software product family.

In (Fig. **1**), the phase 2 is presented as one combined activity: scenario modeling. The purpose of this phase is to define and model a representative set of change scenarios and, if necessary, enhance the existing architectural description with information relevant to the IE evaluation. The scenarios represent possible future needs as regards to integration and extension of a software family. The scenario modeling consists of the following tasks: 1) defining scenarios for integrability and extensibility, 2) selecting appropriate views and patterns for describing architecture, 3) defining matching conditions for interface evaluation, and 4) defining assumptions, architectural constraints and design rationale for each view. The phase results in the description of a software family architecture at the point where all figurative change scenarios have realized. The description includes a complete set of views (e.g. structure, behavior, deployment and development) and selected styles and patterns. The results are used as input to the phase 3.

In the third phase, quality analysts evaluate the integrability and extensibility of the architecture and compare the evaluation results to the defined quality criteria. In quality evaluation, the following activities and techniques are applied: 1) architecture mismatch analysis is done by com-

[3]http://virtual.vtt.fi/qada

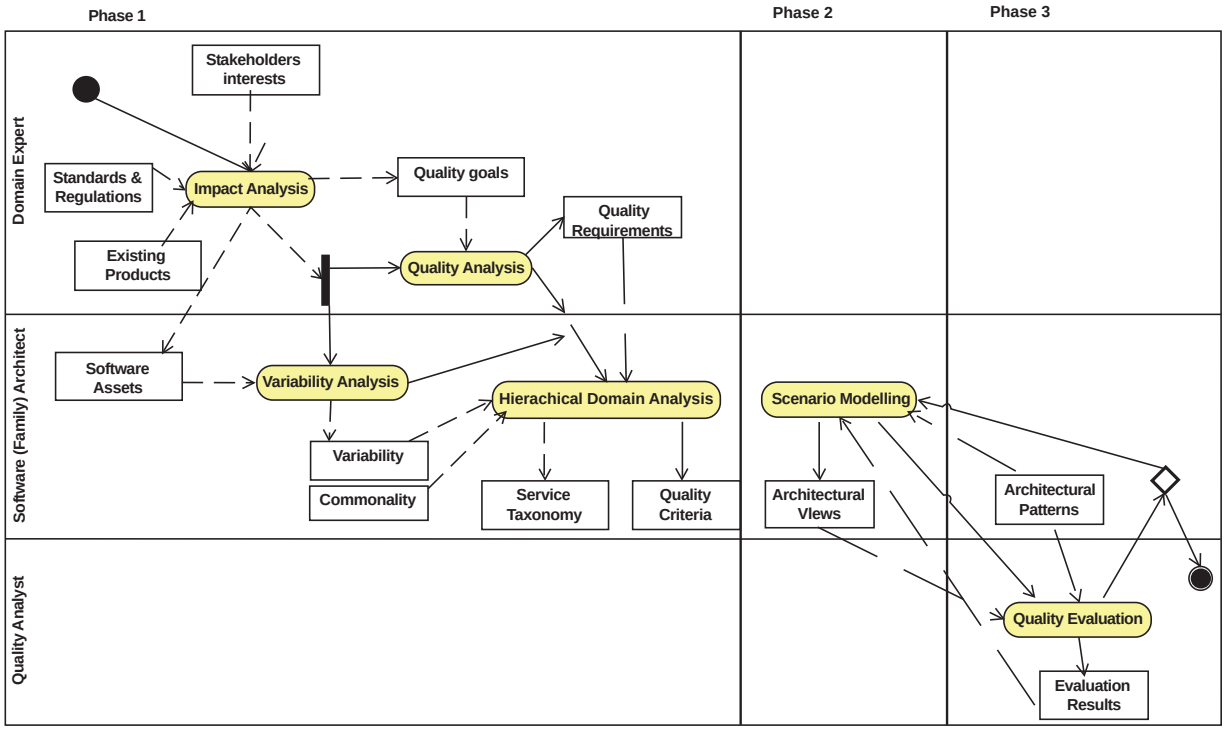

Fig. (1). Overview of the IEE method.

paring component features and used styles and patterns, 2) dependency analysis is used for checking dependencies between interfaces, dependencies between variabilities and dependencies between the binding times (i.e. when variation takes place), 3) extensibility analysis is used to identify in which part the architecture extension points are required and how effectively extensibility patterns are used in the architecture, 4) comparative analysis is applied to compare the evaluation results to the quality criteria, identifying conflicts and making tradeoffs, and finally 6) evaluation results are reported as proposed improvements and identified unsolved problems, which are returned to the software family architects for the next iteration phase.

The quality evaluation is done iteratively and incrementally. First, the quality criteria, which have high importance and affect many parts of the architecture, are evaluated. If these qualities are not met to the requirement, architecture refinement is required and after refinement the quality criteria of high importance are re-evaluated. Secondly, quality criteria of high importance but small impact are evaluated. Third, quality criteria of medium importance for any part of the architecture are taken under evaluation. Last, quality criteria of low importance are checked. The approach allows to focus first on the most important qualities and thereafter to make tradeoffs among the less important quality criteria.

CASE DESCRIPTION: STYLEBASE FOR ECLIPSE TOOL

Overview of the Stylebase for Eclipse Tool

The case study is a model repository tool which is the starting point for the Stylebase for Eclipse[4] product family. Stylebase for Eclipse is a tooling environment for software architects and designers.

The tools are implemented as extensions to the Eclipse[5] platform. Eclipse is a popular open source development environment and a vendor-neutral platform for integrating tools and services. Eclipse has a so called *pure plug-in architecture* [35] which means that there are no core tools in the platform itself and all functionality is implemented as extensions, a.k.a. plug-ins. Each plug-in can define its own *access points* and *extension points*, which allow communication with other plugs in a controlled but loosely coupled manner [36].

Stylebase is a knowledge base which stores information and guidelines of architectural styles, architectural patterns and design patterns in a uniform manner. The idea of maintaining an architectural knowledge base is an important part of QADA methodology [37] discussed previously. The stylebase helps a software architect in selecting styles and patterns, which promote the desired quality goals.

The starting point of the Stylebase for Eclipse product family is a basic tool for browsing and maintaining the stylebase. The tool can be used for both designing and evaluating software architecture. While designing a new architecture model, an architect searches the stylebase according to the desired quality characteristics and selects patterns on that basis. When used for evaluation, an architect detects which patterns have been used in an architecture model and then checks from the stylebase which qualities are associated with these patterns.

The open source implementation of the tool was developed based on previous work [38, 39] and a new open source community was announced in October 2006. By the time of writing this paper (October 2007), five new releases have been issued, the most recent one in September, 2007. The tool has approximately 500 users and there has been more

[4]http://stylebase.sourceforge.net

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

than 2000 downloads on the project website [40]. Five developers have contributed code to the project; three of them are volunteers who come from outside of our research institute.

Software Architecture of the Stylebase for Eclipse

In this section, the current architecture of the model repository tool is described. The following points are discussed: database schema, internal architecture of the tool, outside interface and the selected third-party components and technologies [41].

The tool stores the descriptions of architectural styles and patterns in a relational database. The database was designed in the third normal form (3FN) in order to keep the schema simple and easy to maintain. Fig. (**2**) presents the most essential fields of the tables and illustrates dependencies between them. The abbreviations "PK", "U", and "I" stand for primary key, unique index and index (non-unique), respectively. The table "patterns" contain three large fields. The "model" field contains the data model of a pattern, i.e. structural representation of its components and their interrelations, typically an UML diagram in XML format. "Guide" is a large text field containing the documentation of a pattern, typically stored in HTML format. The field called "picture" stores graphical representation of the pattern in binary format (e.g. jpg/gif).

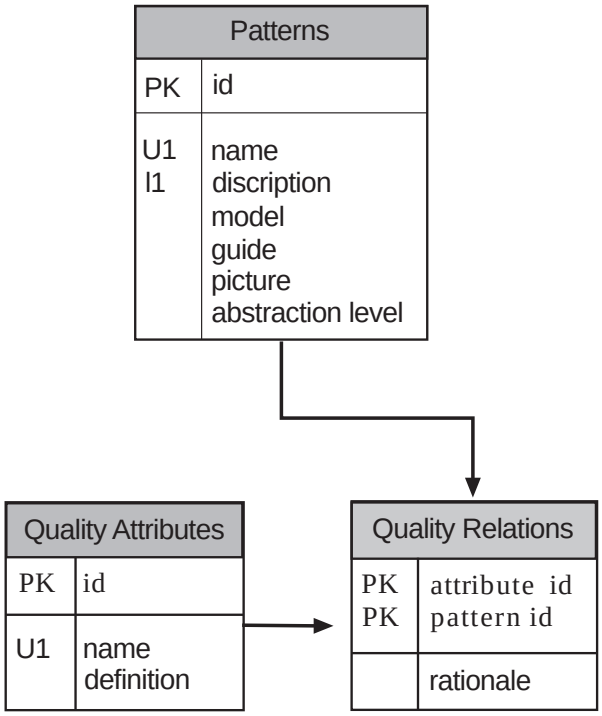

Fig. (2). Database schema for stylebase.

The internal architecture of the model repository tool follows the well-known model-view-controller (MVC) pattern (see e.g. [26]). In the MVC architecture, 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. The pattern supports extensibility [26, 41] and is well-suited for Eclipse plug-in development [41]. In fact, the Eclipse platform itself also follows the model-view-controller architecture [42]. Fig. (**3**) shows how the core plug-in implements a model-view-controller pattern. In order to increase the level of modularity, the architectural subcomponents communicate with each other *via* predefined interfaces (IF).

The view component is responsible for providing the Graphical User Interface (GUI). View attaches to a model and shows model contents on the display. The model notifies the view when model contents have changed and then the view redraws the affected part of the image to reflect these changes. The view also detects GUI events (e.g. mouse click, button press) and sends them to the Controller. A Controller receives events from the View and then commands the Model (Admin) 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. MySQL has been selected as a relational database system and its functionality is hidden behind generic interface. There is also a system component which provides small number of static functions which are accessible from all parts of the Stylebase for Eclipse core plug-in.

The tool also implements access points and extension points which facilitate users to develop downstream plug-ins without touching the source code of the core plug-in. Access points are implemented by building and exporting API (Application Programming Interface) packages. They define a set of functions which developers of other plug-ins may use without detailed knowledge of their internal workings. Extension points are implemented with the extension point mechanism offered by the Eclipse PDE (Plug-in Development Environment). They provide framework for, not only using, but also enhancing the functionality of the core plug-in. The access and extension points provided by the core plug-in are as follows [38]:

Controller Access Point

Provides access to the control component. It allows integrators to associate the controller actions of the Stylebase for Eclipse with the GUI of another plug-in for example to open a dialog for editing quality properties or to check who is locking a pattern.

Model Access Point

Gives access to the Model component. It provides a set of methods for retrieving and updating essential data in the Stylebase.

SQL Database Access Point

Provides direct access to the underlying relational database though SQL query language. It helps in implementing specific functionality not provided by the model interface.

Model 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 extenders to add their own views and/or menu items to the main view of the Stylebase for Eclipse.

In order to provide the desired functionality the core plug-in is integrated with various tools developed by other open source communities. Fig. (**4**) illustrates the selected technologies and their providers as follows [43]. The MySQL database and the associated JDBC (Java Database Connectivity) Driver are provided by a company called MySQL and the open source community it supports. Eclipse Platform and Eclipse Plug-in Development Environment (PDE) are developed by respective communities under the official Eclipse project. The Standard Widgets Toolkit (SWT) is a graphics library for Eclipse plug-ins. The SWT project is managed by the Eclipse platform community, but new widgets originate from the Nebula project which is a source of supplemental SWT widgets and an "incubator" for SWT.

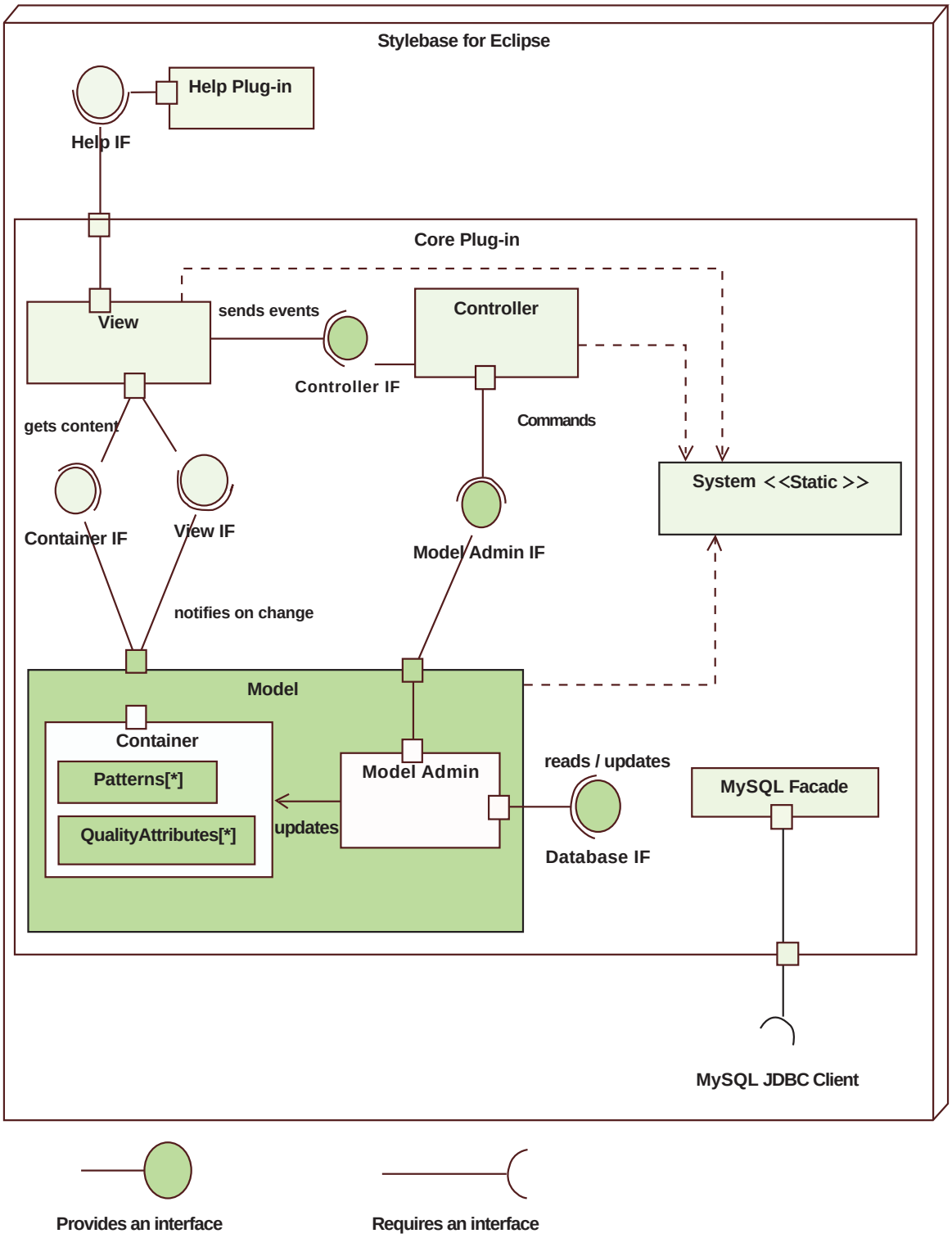

Fig. (3). The current architecture of the Stylebase for Eclipse represented with component diagram.

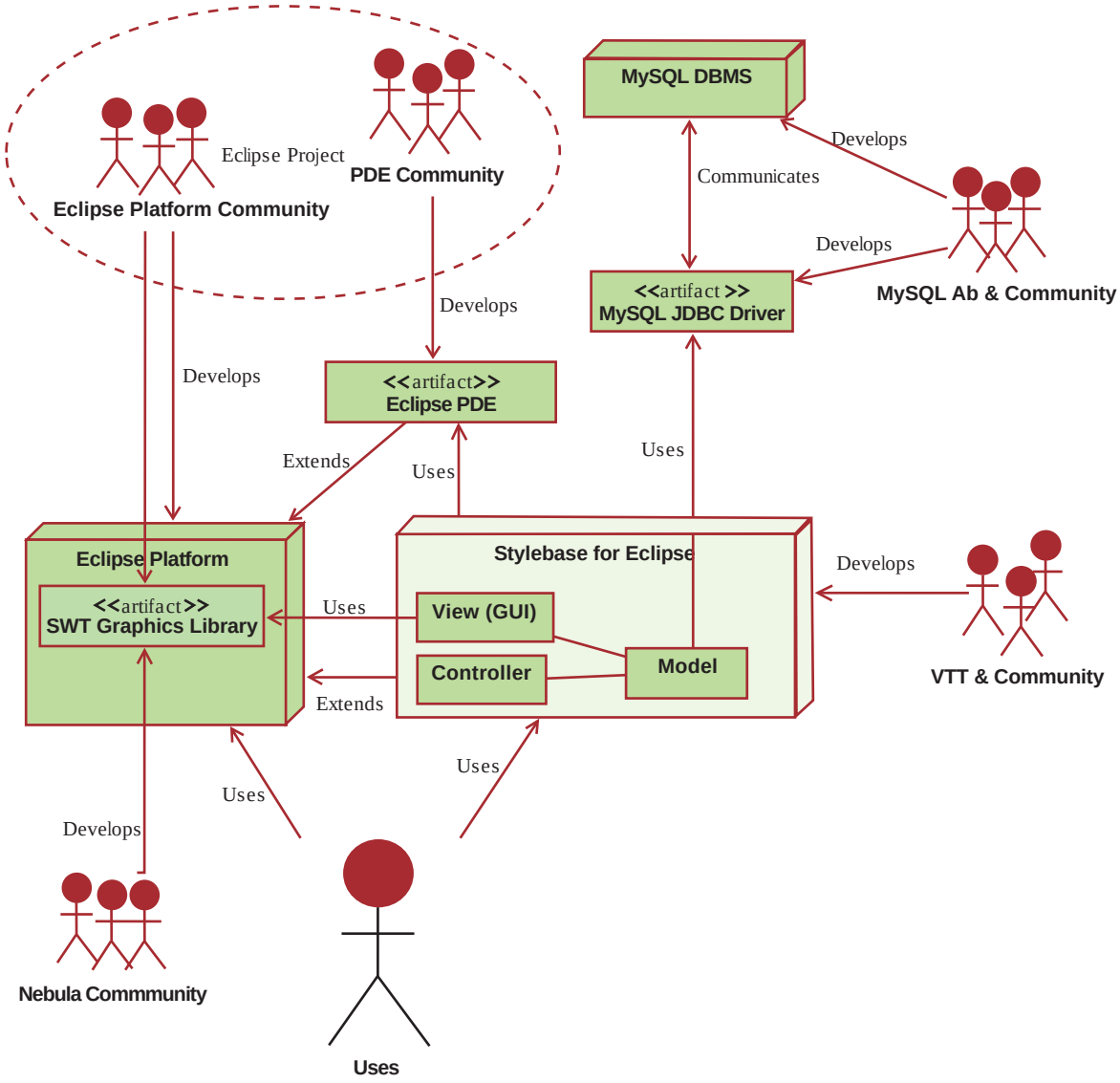

Fig. (4). Integrated components and their providers in Stylebase for Eclipse architecture.

IMPACT ANALYSIS

Stakeholders

IEE quality goal definition is started by identifying stakeholders of the software system. In this example, the stakeholders are identified from software engineering point of view i.e. they describe actions of an individual engineer. The roles are based on what an engineer does with the product and what is his/her relationship to the respective open source community.

From this view point, the stakeholders of the Stylebase for Eclipse tooling environment are as follows:

End User

Utilizes core tools, i.e. the basic functionality of the Stylebase for Eclipse, together with patterns and styles that are distributed with the product.

Advanced End User

Utilizes the extended tooling environment, i.e. both the basic tools and various extensions developed by other stakeholders. This creates new styles and patters and stores them into a local or company-wide database.

Plug-in Extender

Builds custom extensions (i.e. downstream plug-ins) to the Stylebase for Eclipse. Extension may be intended either for personal use or for redistribution.

Plug-in Integrator

Integrates Stylebase for Eclipse programmatically with other plug-ins to improve usability or enhance functionality. Typically acts also as a Plug-in Extender.

Committer

Contributes code to the Stylebase for Eclipse project. Same as Plug-in Integrator or Plug-in Extender, except that

enhancements are published as part of the official Stylebase release. In addition, makes modifications (enhancements, bug fixes) to the basic tools. Experienced committers act as product architects.

Project Leader

Makes final design decisions (as to the development of the core tools) and acts as a product family architect.

When considering the impacts of quality goals from the point of view of business, it is notable that the same customer may act in several software engineering roles. Table **1** presents customer roles (i.e stakeholders defined from business or organizational view point) and their typical relationship to software engineering roles (i.e. stakeholders defined from technical view point).

Quality Goals

Once the stakeholders have been identified, IE quality goals of each stakeholder are elicited. Quality goals to be evaluated concerning *integrability* (I1 – I6) on the architecture level are listed below.

Advanced End User

I1: The product supports a wide range of data models. Consequently, diagrams can be exported and imported to/from heterogeneous modeling tools. Rationale: It is much more convenient to use a familiar modeling tool than purchase a new tool and learn to use it.

Integrator

I2: The product can be programmatically integrated with other plug-ins with minimum development effort. Stylebase plug-ins can be treated as "black box" components if desired. Rationale: If integration requires deep knowledge on the Stylebase architecture or is otherwise time-consuming it will turn away potential volunteer contributors and/or increase expenses of a commercial actor.

I3: Plug-ins in the product family can be developed and tested independently from each other, but still work together as a coherent whole. Many developers can work simultaneously on the product family source code. Rationale: There is no way an open source project could mature unless software architecture supports parallel development [44]. In a modular architecture, developers do not have to learn their way through all the source code before they can start contributing [45].

Committer

I4: The architectural styles of different plug-ins in the product family conform with each other. Rationale: Style conformance decreases the time that developers need to spend in learning the product family architecture. It also eases the integration of plug-ins to some extend.

I5: An existing 3rd party component, which is used by the core product, can be easily substituted with a different one. Rationale: While open source markets evolve, it may be beneficial to switch to a new component or technology which better provides the desired functionality.

I6: Subcomponents of each plug-in can be developed separately from each other, but still operate together as a united whole. Several developers can work simultaneously on the source code of each plug-in. Rationale: Open source development model requires modularity, see rationale for the goal I3.

The quality goals of *extensibility* (E1-E3) evaluation at the architectural level are as follows.

Extender

E1: In addition to patterns, the knowledge base supports to store various types of architectural styles (for example macro, micro and reference architectures plus other, so far undefined, types of styles and patterns). Rationale: Different user groups use the Stylebase

Table 1. Business Stakeholders and their Possible Relation to SW Engineering Stakeholders

Business Stakeholder	Description	Possible relation to SW Engineering Stakeholders
Individual (OSS) Developer	Utilizes design patterns which come with the product. May store his/her own design patterns and idioms for reuse and share them with fellows. May built custom extension for personal use or even contributes to global release as a committer.	End User Advanced End User Plug-in Extender Committer
Open Source Integrator	Develops an integrated system and uses selected modules of the Stylebase for Eclipse product family as part of it. May contribute to the global Stylebase release.	Plug-in Integrator Plug-in Extender Committer
Small Company as Utilizer	Stores both design patterns and architectural patterns for reuse. The knowledge is shared in a local or distributed development team.	Advanced End User
Big Company as Utilizer	Same as the small company, but can spent more effort in deploying the product, e.g. by integrating it with other tools of their choice.	Advanced End User Plug-in Integrator

plug-in for different ends and thus need to store different types of data.

E2: Downstream plug-ins can be built with minimum development effort and without touching the source code of the core plug-in. Rationale: Modularity facilitates parallel development and allows the core product to stabilize [45]. If building extension is laborious, it will turn away potential volunteer contributors and/or increase expenses of business actors. Furthermore, the architecture of the Eclipse framework is based on the idea of having piles of plug-ins built on top of each other [35].

Committer

E3: A new feature can be added to the core plug-in with minimum development effort and without changing the existing architectural style. The architecture remains simple and easy to learn. Rationale: This saves development efforts and helps to keep the product evolving.

QUALITY AND VARIABILITY ANALYSIS

The purpose of quality analysis is to separate quality concerns related to business, constraints and functionality. The purpose of the variability analysis is to define the requirements that vary on the business domains or stakeholders and to separate commonality and specialty of variations in domains. The variability analysis is then continued by considering dependencies of the IE goals [34].

In Table **2,** quality goals are categorized and the importance of the quality goals is estimated from the view point of each stakeholder. Fig. (**5**) represents the Strategic Dependency Model [46], which describes the dependencies between quality goals, functional domains and stakeholders. The circle corresponds to stakeholder, rectangles to the required functionality and ellipses to the IE requirements. Arrows indicate dependencies (e.g. a plug-in integrator is dependent on quality goal I2 and relies on committer stakeholders to implement that goal).

Table 2. Variability in Quality Goals Per Each Stakeholder

Category	Quality goals	Stakeholders	Priority
Plug-in Integrability	I1: Support for heterogeneous data models and modeling tools	Integrator, Committer Advanced User, Extender Passive User	Very High High Medium
	I2: Integrability with other plug-ins as "black box" components.	Integrator Committer	Very High Low
	I3: Support for parallel development of plug-ins	Committer, Integrator	Very High
	I4: Style conformance of product family architecture	Committer Integrator	Medium Low
Component integrability	I5: Substitutability of third party subcomponents	Committer Integrator, Extender	Very High Medium
	I6: Support for parallel development of subcomponents	Committer Integrator, Extender	Very High Low
Plug-in Extensibility	E1: Capacity to store entirely new types of data models	Committer, Extender Integrator Advanced User	Very High Medium Low
	E2: Support for building downstream plug-ins	Extender, Integrator Committer	Very high Low
	E3: Expandable plug-in architecture	Committer Extender, Integrator	Very High Medium

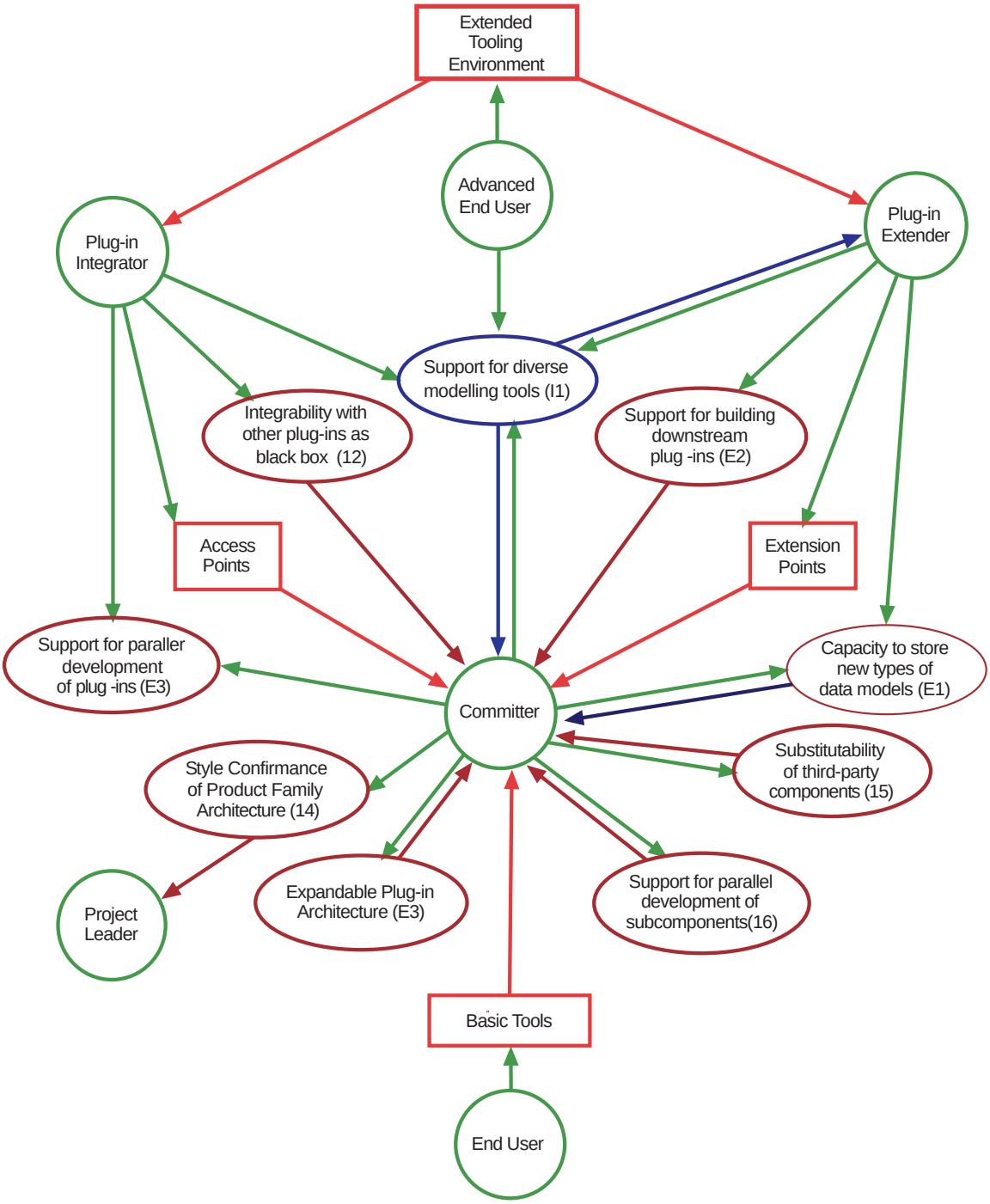

Fig. (5). Dependency model of IE requirements.

HIERARCICAL DOMAIN ANALYSIS

When IE quality categories have been defined and their prioritization has been done, the hierarchical domain analysis is used to map the quality goals into to domains, sub-domains, components or services of an application.

Table **3** presents the domain specific IE goals, i.e. goals that apply to a certain domain/domains of the core plug-in. These goals also apply to respective domains of dependent-plug-ins which are later added to the Stylebase for Eclipse product family.

Three of the previously defined goals are not mentioned in the Table **3** because they cannot be assigned to a particular domain of the core plug-in. These goals must be implemented in the entire architecture as presented in Table **4**.

The purpose of the hierarchal domain analysis is to identify quality goals whose influence on the family architecture is the largest. Goals that are not domain specific (I3, I4 and E3) have the highest impact and are capable of breaking the entire software architecture in the case of conflict. Goals such as E1 and I1, which apply to basic functions on storing

Table 3. Mapping of the IE Goals into Domains/Components of the Core Plug-In

Functional Domain	Component	Responsibility of Component	Domain-Specific Goals
Data Management	Database Schema	Store architecture and design patterns	I1 & E1: Support storing any data model independently of its internal structure
	MySQL Facade	Provide a simplified library for reading and writing to the remove database	I5: Encapsulate MySQL specific functionality and hide it beyond a generic interface
	Model	Provide a run-time storage object and an interface for managing the pattern database	I1: Support managing heterogeneous data models I6: Communicate with View (refresh requests) *via* a pre-defined interface
User Interface	View	Provide a graphical user interface (GUI) for browsing and updating the patter repository.	I6: Communicate with Controller *via* a pre-defined interface
	Controller	Map input signals from the user interface into the application response.	I6: Communicate with Model *via* a pre-defined interface
Programming Interface	Access Points	Provide programming interface which allows other plug-ins to calls the functions of Stylebase for Eclipse	I2: Provide access points to MySQL Façade, Model and Controller components
	Extension Points	Provide extensible programming interface which allows other plug-ins to enhance the functionality of the Stylebase for Eclipse	E2: Provide extension points for adding new models and GUI elements
Other	System	Provide a small library of static system functions accessed from different parts of the core plug-in	I2: Support lazy initialization of dependent plug-ins where appropriate

Table 4. Goals that Apply to the Entire Architecture

Scope	Goals
Product Family Architecture	**13**: Product family architecture must consist of several loosely couple plug-ins rather than few large ones. **14**: Architectural style selected for each plug-in must confirm with styles of other plug-ins in the product family.
Plug-in Architecture	**E3**: Each plug-in must implement an architectural style that supports extensibility.

Table 5. Strategy/Criteria for Each Quality Goal

Goal	Solution Strategy / Technical Criteria
I1	• generic database fields • reusable, type-safe data structures
I2	• comprehensive and easy-to-use programming interface (API) for each plug-in • selecting architectural patterns that support integrability
I3	• low coupling and dynamic binding of plug-ins • localizing dependency on other plug-ins • small plug-in size
I4	• using only few styles and patterns product family wide, providing clear design rationales for them
I5	• localizing dependency on third-party subcomponents • selecting design patterns which support integrability
I6	• maximum cohesion of subcomponents • low coupling of subcomponents, inter-component communication *via* controlled interfaces • small component size
E1	• generic database fields • reusable, type-safe data structures
E2	• implementing comprehensive set of extension points
E3	• selecting architectural styles and design patterns which support extensibility

and retrieving data, also have a significant influence. On the other hand, we can notice that goals E2 and I2, for example, apply to limited points in the architecture. In case of conflicts, there would be no need to re-design the whole software architecture and therefore we can state that these goals have smaller influence.

PRIORITIZING QUALITY GOALS AND DEFINING CRITERIA

The hierarchical domain analysis and variability analysis are used to prioritize the quality goals. In this case study, the goal I1 stood out as the most important from the view point of stakeholders (see Table **2**). Hierarchical domain analysis (see Table **3** and Table **4**) ranked the goals I3, I4 and E3 as the most crucial.

At the end of this phase, one needs to decide which quality goals are selected for evaluation. In commercial projects,

cost factors have obviously impact on how many quality goals are considered important enough to be evaluated. The goals can be grouped into categories according to their significance and the evaluation should always start from the most important ones.

In this case study, the target system is relatively small and the list of quality goals did not grow too extensive. It was therefore decided to evaluate all quality goals. Table **5** summarizes solution strategies for each quality goal. The strategies constitute the technical evaluation criteria and shall be discussed in detail later on.

SCENARIO MODELING

As explained previously, the IEE method is a scenario based evaluation method. The scenarios are created by considering possible ways in which the product might be integrated or extended in the future. Since it is clearly infeasible to identify and include all possible scenarios, we defined three scenarios which represent the following categories: (1) replacing existing components, (2) adding new features by building a dependent plug-in and (3) adding new functionality by integrating with another plug-in. This section describes three scenarios which cover the evaluation of the nine quality goals defined previously.

Scenario 1: Replacing a Third Party Component with Variants

The current version of the Stylebase for Eclipse relies on MySQL database for storing patterns. Therefore all users are required to install MySQL which, according to feedback from the community, makes the installation process too time-consuming and error-prone. In this scenario, a new variation point is introduced into the Stylebase for Eclipse product family. Product variant A will be based on a MySQL while product variant B shall be based on a relational database product such as HSQLDB, which can be embedded into a Java application in a way invisible to the end-user. Variant C shall be based on a file database. All the products shall offer essentially the same functionality.

As previously illustrated by the current architecture of Stylebase for Eclipse (see Fig. **3**), the Model component is divided into two parts: the Model Administrator and the Container. The primary function of the Model Administrator is to update the Container and construct SQL strings. The scenario requires that functionality common for all products (i.e updating the Container) is detached from the Model Administrator. Once this is done, one can make one Model Administrator for constructing SQL strings (for products A and B) and another for constructing XML queries (for product C).

In the Eclipse framework, functional variability is typically implemented by splitting one plug-in into several dependent plug-ins [47]. The plug-ins are then bunched together to create the desired functionality for each version/product. Fig. (**6**) illustrates how plug-ins should be combined to create product variants A, B and C as defined previously. The division contributes to modularity as required by the goal I3.

IE quality goals require a comprehensive set of access and extension points, which are the only points of communication between plug-ins. Therefore, two new extension points must be added: (1) an extension for adding new "Model Administrators" to the Core Plug-in and (2) an extension point for adding support to new SQL clients[6] to the SQL DB Plug-in.

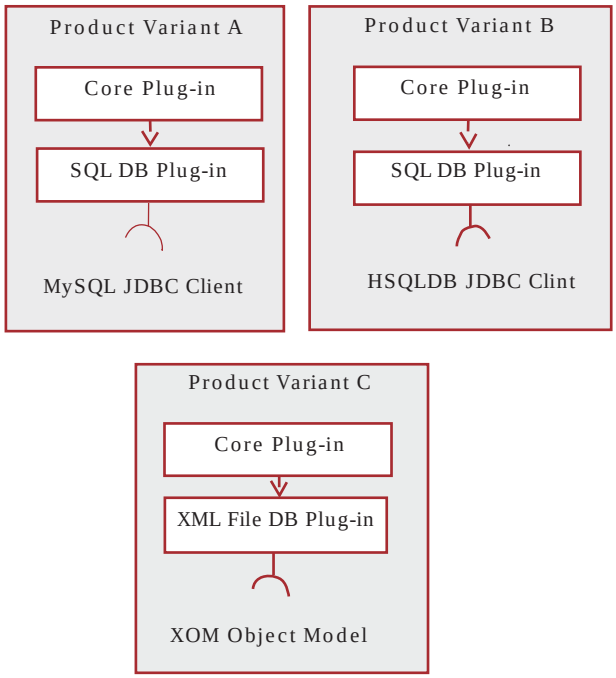

Fig. (6). Composition of each product variant in the Stylebase product family.

A new Model Administrator class can implement the same interface as the old class and thereby conforms perfectly to the existing architectural style. Similarly, it is easy to add support for a new relational database because functionality specific to the MySQL Client is encapsulated in a Façade [27] class which implements a generic interface. The Façade can be quickly transformed into an Adapter [27]. However, there is more than just adapting interfaces. While vast majority of used SQL statements are such that they can be understood by any relational database product, there may be exceptions. For example, the syntax of an SQL clause which performs full text searches on MySQL is not supported by other database products. The issue needs to be addressed, for example by sub classing the Model Administrator inside the SQL DB Plug-in.

Fig. (**7**) presents the new structure of the product family architecture. The plug-ins communicate with each other *via* access points (AP) and extension points (EP). The new components added in this scenario are highlighted with red color while the yellow color identifies the old components which have been modified and/or relocated. The green components remain unchanged in this scenario.

Scenario 2: Building a Custom Extension to Enhance the Functionality of an Existing Service

In this scenario, the core capabilities of the Stylebase for Eclipse are enhanced. The tool is made capable of storing and browsing *reference architectures* (see e.g. [23]) in addition to design patterns and architectural patterns. Further-

[6] A third-party plug-in to support Oracle client is included in Figure 7 as an example on how the SQL Client Extension point may be used.

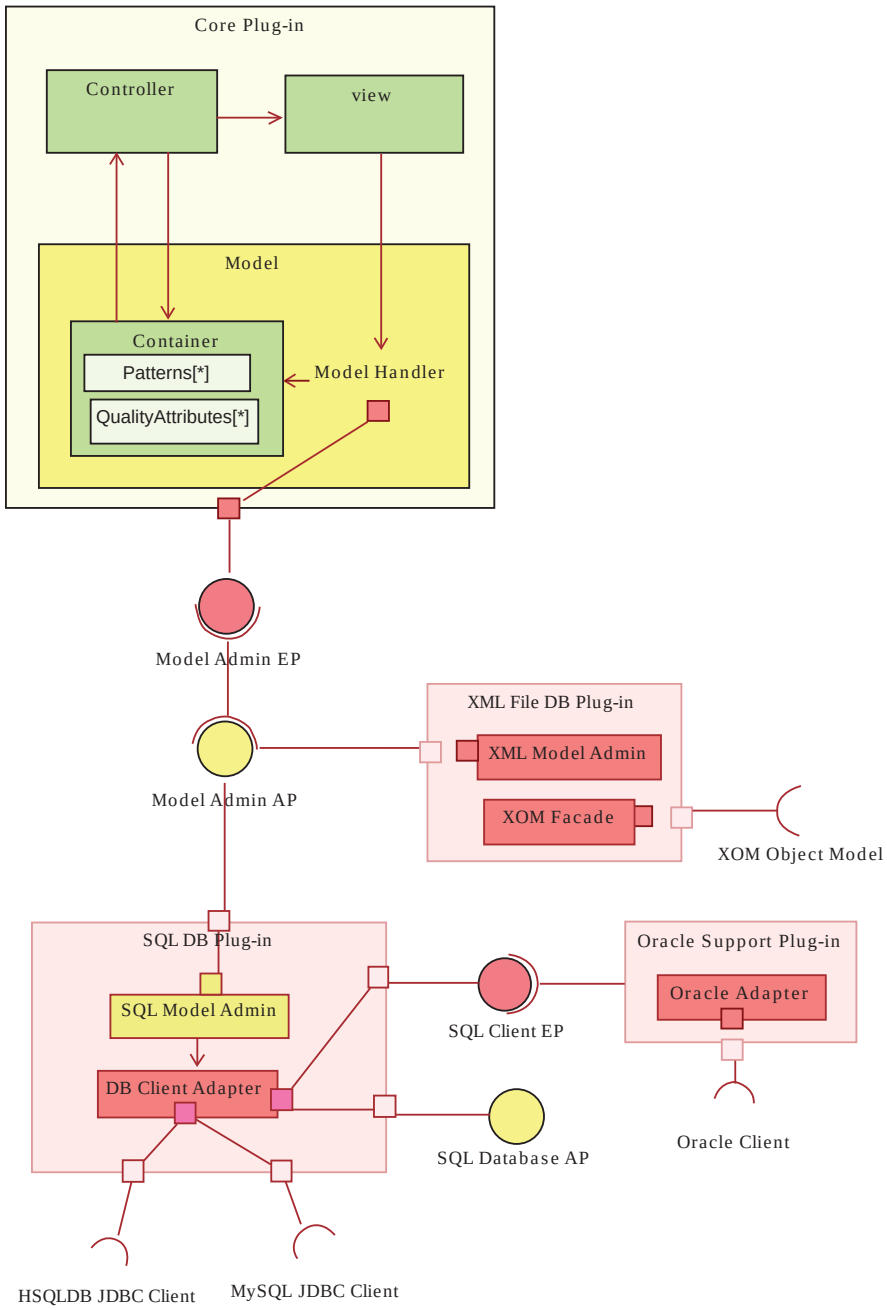

Fig. (7). Replacing existing component with optional variants.

more, DBMS based product variants A and B (see scenario 1) shall store relationships between architectural models and perform searches on them. For example, a user may want to list architecture patterns used in a particular reference architecture or search for design patterns associated with a particular architecture pattern.

Firstly, possible changes to the database schema need to be examined. The existing database schema is illustrated in (Fig. **2**) (see section "Software Architecture of the Stylebase for Eclipse"). Because data model of each pattern is stored in XML format into a large text field, the database schema does not anyhow restrict the type of model being stored. The "patterns" table can thus store reference architectures without any modifications; it could be obviously renamed as "styles" to describe the new content. In addition, a new table is needed to store the relationships between styles and patters. Fig. (**8**) illustrates the database schema after a new table called "Style Relations" has been added.

The structure of the Container component in the core plug-in reflects to that of the database schema. It only contains basic fields – such as ID, name and description – which are required by patterns and reference architectures alike. Large text and binary fields are not stored in the run-time container but fetched from database per request. Reference architectures can thus be stored into the same container and managed by the same model administrator classes as patterns.

In the previous scenario, SQL Plug-in and XML File DB Plug-in were assumed to provide the same operations and

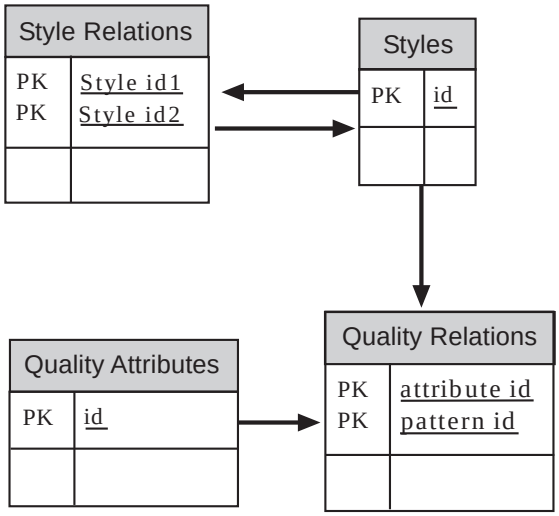

Fig. (8). New table is added to the database schema.

thus implement the same interface. This scenario introduces new functional variability as only SQL based product should detect relationships between models. There are two alternative solutions. Firstly, the SQL/XML plug-ins can implement different interfaces and, in this case, an Adapter class should be added to the core plug-in. Secondly, new operations can be encapsulated into a separate plug-in which enhances the functionality of the SQL plug-in. The last mentioned solution was selected, because this scenario also introduces enhancements to the view and controller components. Considering that not all products in the product family use the features, it is not desirable to include them into the core plug-in.

The new plug-in shall also follow the Model-View-Controller architecture. The plug-in needs a view component for implementing new GUI elements and relies on the GUI extension point for attaching these enhancements to the core plug-in. The Model component of a new plug-in will store and manage the data on style relations. It provides content for the View component, together with the Model of the core plug-in. The dependent plug-in also implements its own SQL Model Admin which enhances the one of the SQL Plug-in. Fig. (**9**) illustrates the structure of the new plug-in and how it communicates with other plug-ins (UML2 component diagram). The red color highlights the components added in this scenario.

Scenario 3: Adding New Functionality by Integrating with Another Plug-In

In order to provide more benefits to the end-users, patterns and styles stored in the stylebase should be easily exported and used in combination with other plug-ins. In this scenario, we integrate Stylebase for Eclipse with the Eclipse Modeling Framework.

The first consideration is the interoperability of data models. The Eclipse Modeling Framework relies on an Eclipse UML data model called “eCore” which is part of the Eclipse UML2 metamodel framework. Practically, every UML modeling tool on the market produces a different type

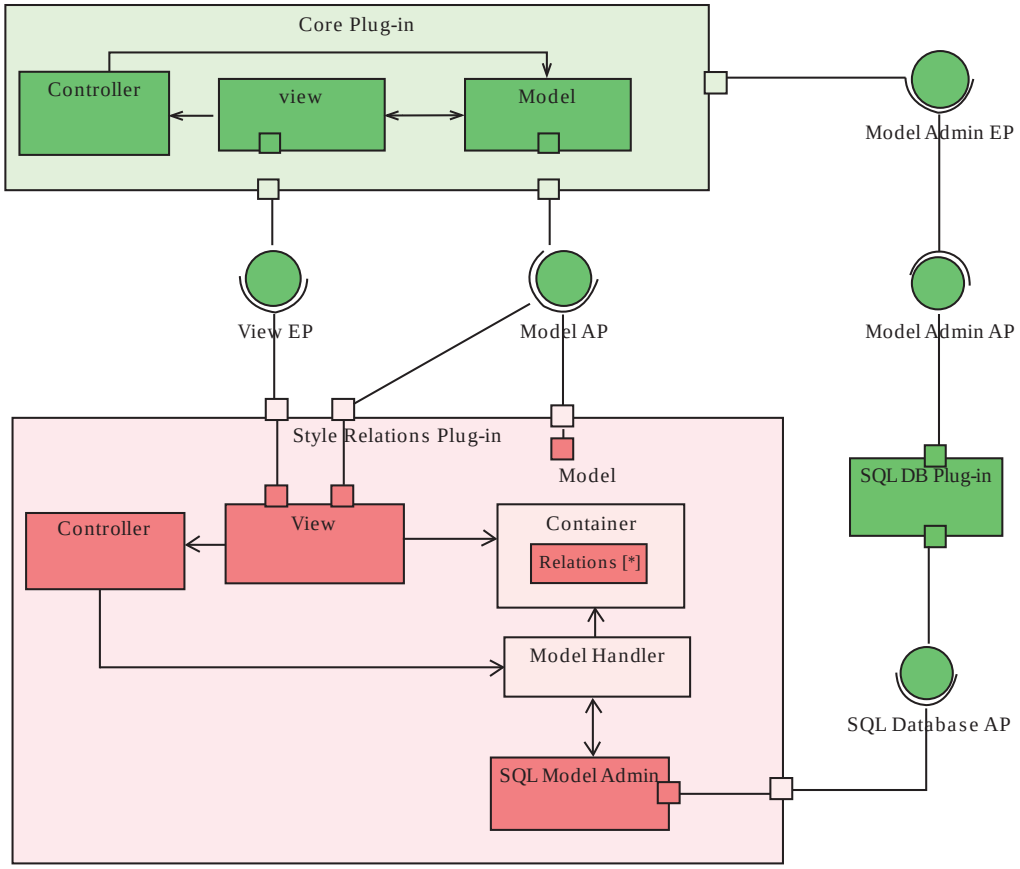

Fig. (9). Adding new features by building a dependent plug-in.

of data model. There is no widely accepted standard which would be specific enough to ensure compliance. A few modeling tools (e.g. Rational Rose and Topcased) can generate Ecore models, but many others (e.g. ArgoUML, Start UML) cannot. Therefore we first have to implement an additional component which can perform conversion from heterogeneous data models to the "eCore" model and back.

The converter component is not only required by EMF integration but by the whole product family. The converter is needed, for example, to enable the users of heterogeneous modeling tools to share the same knowledge base. Model conversion can thus be regarded as one of the most essential components of the Stylebase product family. Despite of its central role, we decided to model the converter as a separate plug-in. This would facilitate parallel development and allow the converter to be used in combination with any other plug-in, which, in turn, could attract other communities to share the development effort.

In addition to the model conversion, Stylebase for Eclipse needs to open or create an EMF project and launch the EMF Editor. This is done by using a customizable wizard provided by the programming interface of EMF. If the files are edited with the EMF, the Stylebase for Eclipse should be capable of loading updated patterns back to the knowledge base. This is done by using Eclipse resource handling capabilities together with a component called EMF Model Exporter.

Fig. (**10**) illustrates the new structure of the product family architecture. The components implemented in this scenario are highlighted with red color. The support for EMF is again encapsulated into a separate plug-in. The EMF Support Plug-in has its own View and Control components, but relies on the Model component of the core plug-in. The View component implements one or two menu items which extend the context menu of the main view. The Controller retrieves desired patterns from the Model component and passes them to the EMF Adapter. The EMF Adapter uses the Conversion

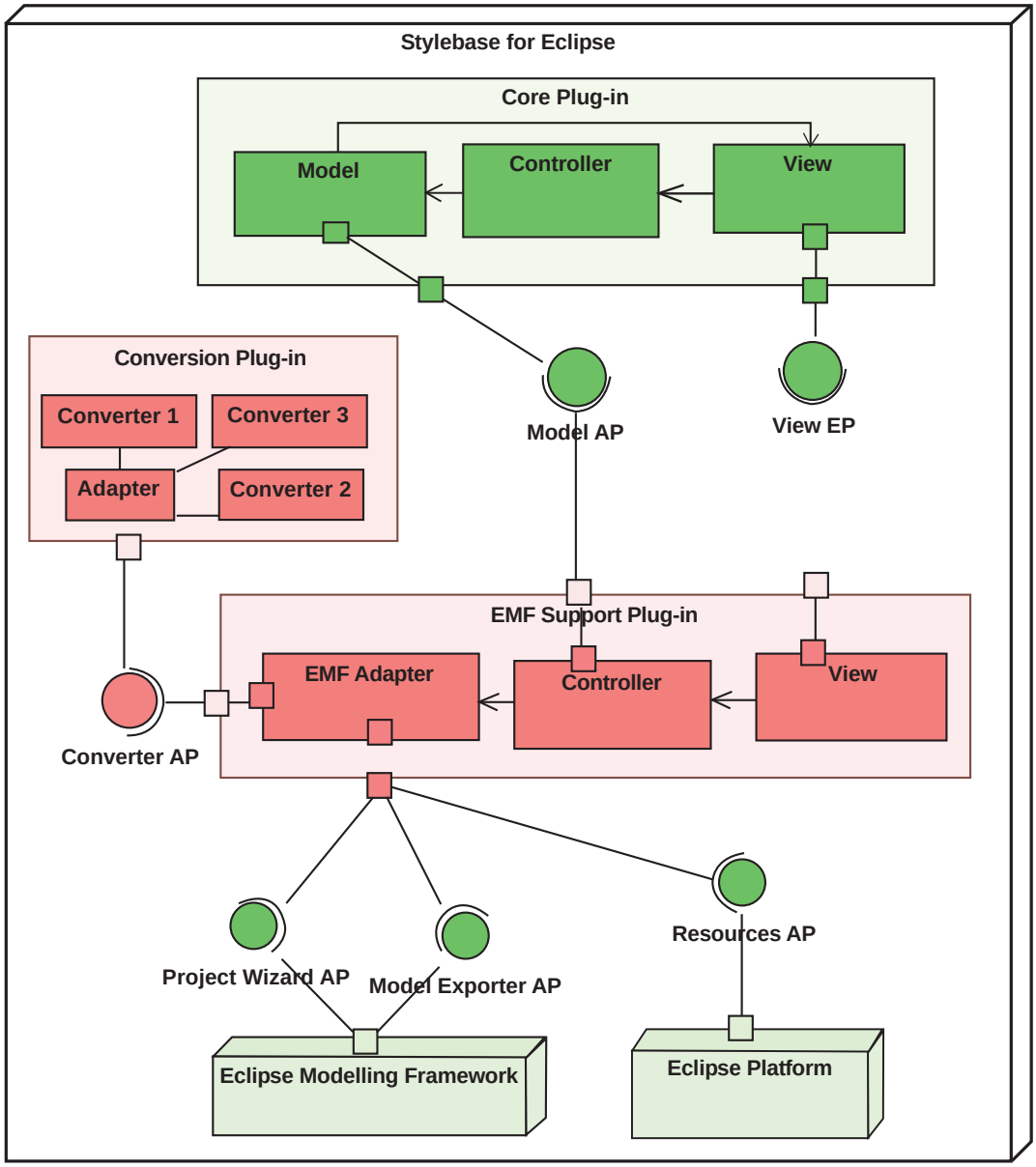

Fig. (10). Adding new functionality by integrating with EMF.

Plug-in to translate pattern's data model into eCore format and then sends it to the Eclipse Modeling Framework for processing. The Conversion Plug-in contains a number of actual converters, which translate an Ecore data model into a format understood by a particular UML tool and then back. There is one converter for each model type. Because individual converters may come from a third party, one cannot count on them to implement the same interface and, consequently, an Adapter is needed.

QUALITY EVALUATION

Once we have modeled the architecture to support the IE evaluation, we must evaluate how the quality goals are met in the architecture. The evaluation is based on the scenario modeling illustrated above. The format used in reporting the evaluation results is specified as following:

- Identifier and name of the quality goal
- Goal definition: What is the concrete definition for this quality goal?
- Strategy: How do we plan to achieve the goal?
- Evaluation on how current architecture supports the quality goal
- Status: How well is the goal achieved? (fully/mostly/ partially/not achieved)

I1: Support for Heterogeneous Modeling Tools

Goal Definition

Stylebase for Eclipse can be used in combination with heterogeneous modeling tools.

Strategy

Database stores the XML representation of each data model in one last text field, thus being unaware of its internal structure. Runtime data structures are equally generic.

Evaluation

In the current version of the Stylebase for Eclipse, the support for heterogeneous modeling tools is implemented solely on the database level. In the 2nd scenario we saw that database schema and other memory objects are ignorant to the internal structure of a data model and, consequently, can store the output of any modeling tool. This enables users to use the Stylebase for Eclipse with a modeling tool of their choice. However, the 3rd scenario disclosed that the originally defined implementation strategy is not fully sufficient. Users wishing to share data have to agree on the use of the same modeling tool. This reduces integrability with other plug-ins and decreases the usefulness of the global knowledge base. In order to fully achieve this goal, architecture must provide framework for building converters.

Status

The goal is partially achieved.

I2: Integrability with Third-Party Plug-Ins

Goal Definition

Plug-ins in the product family can be integrated with third-party plug-ins with minimum development effort and as "blax box" components.

Strategy

Each plug-in must implement comprehensive and learnable programming interface, i.e. access points.

Evaluation

The current set of access points is sufficient for integrating the core plug-in with other plug-ins in the modeled scenarios. The core plug-in implements three access points: one for sending GUI events programmatically to Controller, one for reading and updating data in the Model and one for SQL-level access to stylebase. Access points contain a comprehensive set of functions because they are derived from internal component interfaces illustrated in (Fig. **3**). Even though the Controller access point is not used in any scenario, it proves to be useful when one wants to launch already built functionality from the user interface of another plug-in. In fact, this is exactly how the Project Wizard access point of the Eclipse Modelling Framework is used in the 3rd scenario.

Status

The goal is fully achieved.

I3: Support for Parallel Development of Plug-Ins

Goal Definition

Product family architecture consists of several plug-ins which can be developed independently of each other.

Strategy

Plug-ins are reasonably small in size. They should communicate only *via* the pre-defined access points and extension points, thus enabling dynamic binding.

Evaluation

The current version of the Stylebase for Eclipse contains only two plug-ins. The core plug-in uses an access point to utilize the functionality of the help plug-in as illustrated in (Fig. **3**). Even though the current core plug-in seems reasonably small, the 1st scenario proved that it might become necessary to split it into even smaller parts (see the evaluation of the goal I8). However, the current plug-in composition seems practical for the time being. The requirement was also taken into account while modeling the scenarios. Therefore, several new plug-ins were introduced and access and extension points were defined for communication. The goal is easy to achieve because Eclipse supports loose coupling and dynamic binding of plug-ins by design.

Status

The goal is fully achieved.

I4: Style Conformance of Product Family Architecture

Goal Definition

The architectural styles of plug-ins in the product family conform to each other. The product family architecture is simple and learnable.

Strategy

Only one or few styles are used product family wide, and the main patterns are used several times. Rationale for used patterns and styles is presented.

Evaluation

The Model-View-Controller has been selected as the main architectural style of the core plug-in for reasons explained previously (see section "Software Architecture of the Stylebase for Eclipse"). The modifications made in the scenarios do not force changes in the selected style. In fact, scenarios 2 and 3 demonstrate that dependent plug-ins can adapt the same architectural style and the style conformance in the product family is thereby maintained. The Adapter and Facade patterns, which efficiently support integrability and extensibility, can be used product family wide.

Status

The goal is fully achieved.

I5: Substitutability of the Third Party Subcomponents

Goal Definition

Third party subcomponents can be replaced with minimum development effort / without changing the existing architectural style.

Strategy

Dependencies on third-party subcomponents are localized. Design patterns which support integrability are used.

Evaluation

The third party subcomponent – MySQL database – is replaced in the first scenario. MySQL can be replaced with another relational database with relatively small development effort. This is because MySQL specific functionality is encapsulated into one class (Facade) and hidden beyond a generic interface. Facade can easily be transformed into an Adapter to provide support for other types of SQL clients. In practice, there is a problem in the fact that the syntax of some SQL clauses may not be supported by all database products. The current version does not implement any structure for solving the issue. The scenario reveals another problem when replacing MySQL database with a file database. In the current version, the Model Administrator class takes care of both updating the Container and constructing SQL statements. It is therefore an unnecessarily hard work to divide it into two plug-ins as in scenario 1. In both cases, the primary architectural style remains intact.

Status

The goal is partially achieved.

I6: Support for Parallel Development of Subcomponents

Goal Definition

Each plug-in consists of several subcomponents which can be developed independently from each other, but still operate as a coherent whole.

Strategy

Subcomponents are reasonably small in size and communicate *via* controlled interfaces.

Evaluation

As illustrated by (Fig. **3**), all components of the core plug-in communicate *via* controlled interfaces. As long as the plug-in size is kept small (see goal I3), the individual subcomponents are not likely to grow too large either. In all scenarios, new features are implemented as dependent plug-ins and therefore the size of core components remains small and parallel development is facilitated. However, the modeling of the 2nd scenario indicates that - unless a new plug-in is introduced – adding a new feature would affect each component of the core plug-in: the Model, View and Controller. In fact, most new features would include a new user interface element (View), reaction to the input from the user interface (Controller) and modifying information in a new way (Model). Therefore a very small enhancement – such as adding a button for loading quality attributes from text file – would require updating all three components. In the Model-View-Controller architecture, simultaneous development of main components may not be straightforward, which highlights the importance of the goal I3.

Status

The goal is mostly achieved.

E1: Capacity to Store Completely New Types of Data Models

Goal Definition

In addition to patterns, the knowledge base supports storing of various types of architectural styles, e.g. macro, micro and reference architectures plus other, so far undefined, types of styles and patterns.

Strategy

Database stores the XML representation of each data model in one large text field, thus being unaware of its structure. Runtime memory structures are equally generic.

Evaluation

This goal was already included in the initial requirement specification of the Stylebase for Eclipse [43]. The strategy for the goal is exactly the same as the one used to ensure support for heterogeneous data models in the goal I3. Scenario 3 demonstrates that database and other memory objects do not anyhow dictate on the structure of a model being stored. This ensures that the database can store new types of patterns and styles without any changes to the database schema.

Status

The goal is fully achieved.

E2: Support for Building Downstream Plug-Ins

Goal Definition

Dependent plug-ins can be built without touching the source code of the core plug-in / without detailed knowledge of its internal workings.

Strategy

Each plug-in implements a comprehensive set of extension points for downstream plug-ins to use.

Evaluation

The scenarios demonstrate that the current set of extension points is not optimal. The core plug-in implements two

extension points: one is for enhancing user interface with new views or menu items and another one is for adding new types of Model objects. The View Extension Point is used in both 2nd and 3rd scenario. The 2nd scenario suggests that the Model Extension Point may not be needed because the current memory object is flexible enough to store practically any kind of data model. In turn, the first scenario indicates that two new extension points should be added: one for adding support to heterogeneous SQL clients and another one adding new types of Model Administrators.

Status

The goal is partially achieved.

E3: Expandable Plug-In Architecture

Goal Definition

Architecture of each plug-in can be extended with minimum development effort without changing the existing architectural style.

Strategy

Architectural styles and design practices which support extensibility are used.

Evaluation

Scenarios demonstrate that there is no need to change the selected architectural style when adding new features or services. On the contrary, the Model-View-Controller (MVC) architecture promotes extensibility. In MVC architecture, same views may be used to show data from different models. For example, in the 2nd scenario, the Style Relations Plug-in can reuse views of the core plug-in for showing its own information. Because MVC allows multiple representations of the same information, Style Relations Plug-in can also implement views which show content from the Model of the core plug-in. Furthermore, dependent plug-ins, which adapt the same architectural style, do not need to implement all three components – Model, View and Controller – independently. They can rely on one or more respective components of a core plug-in as in the 3rd scenario.

On the design level, Facade pattern is used to reduce dependencies of outside code on the inner workings of a component or software library. When the wrapper needs to support polymorphic behavior, Facade can easily be transformed to an Adapter. Expanding of the architecture is also eased by that fact that subcomponents communicate *via* controlled interfaces. For example, we noticed in the 1st scenario that a new type of Model Administrator class can be hidden behind an existing interface which saves development effort.

Status

The goal is fully achieved.

Summary of Evaluation Results

The evaluation revealed that five of the nine quality goals were fully implemented in the architecture. One quality was covered mostly and three were covered only partly by the architecture. Table 6 summarizes the results of the IE evaluation and shows which of the modeled scenarios where used in the evaluation of each quality goal.

Improvement Suggestions

The first scenario demonstrated that two new extension points should be added. Firstly, there should be an extension point for adding support to new types of SQL clients. This is because some users may want to integrate the Stylebase for Eclipse with their own relational database system rather than install MySQL. An extension point for adding a new type of Model Administrator class would also be desirable – and, in fact, compulsorily, if the primary Model Administrator is relocated into a separate plug-in as in the scenario 1. It was also demonstrated that the Model Administrator needs to be divided into two separate parts. One should be responsible for building SQL statements and another one for updating the Container component. In the future, the product is likely to need to support other types of databases than SQL ones and therefore it is necessary to separate the SQL statement construction from the core functionality. The reconstruction of the Model Administrator is important also because SQL clauses may be specific to one particular database product.

Table 6. Results of IE Evaluation Summarized

Quality Goal	Scenarios			Status
	1	2	3	
I1: Support for heterogeneous modeling tools	X	X	X	Partially
I2: Integrability with other plug-ins as “black box”	X	X	X	Fully
I3: Support for parallel development of plug-ins	X	-	-	Fully
I4: Style conformance of product family architecture	X	X	X	Fully
I5: Substitutability of third party subcomponents	X	-	-	Partially
I6: Support for parallel development of subcomponents	-	X	-	Mostly
E1: Capacity to store entirely new types of data models	-	X	-	Fully
E2: Support for building downstream plug-ins	X	X	X	Partially
E3: Easily expandable plug-in architecture	X	X	X	Fully

As highlighted by the 3rd scenario, it is evident that product must implement a framework for building converters which translate a data model generated by one modeling tool into a format understood by another tool. While the community does not have resources to implement converters from every tool to another, a good framework should encourage users to implement their own and hopefully contribute them back to the community. In order to maximize outside participation, the converter component must be combinable with any Eclipse plug-in, not just Stylebase for Eclipse.

DISCUSSION AND CONCLUSIONS

Integrability and extensibility might be the most important evolution qualities of future product families. This is because future software systems will mainly be based on collaborative software development, where several partners are developing complex products together, each of them focusing on their own competencies and the parts of the systems that provide profit and added value to the company. Needs for integrability of product families arise from the extended use of the third party components and services. Extensibility is required because of rapidly changing markets and the need for innovative new products in order to compete successfully in the market. The significance of extensibility and integrability is especially highlighted in open source software development because of its collaborative nature and de-centralized requirements engineering process.

The IEE method (together with the QRF method) covers all the phases of software family engineering that are related to architecture development: specification of the quality goals, transferring quality goals to architectural models, and evaluating how the defined quality goals are achieved by the architecture. Each phase comprises a set of steps and a number of specific activities that have to be carried out during the phase. In this paper, we addressed how to apply the IEE method in the context of open source software.

The results of applying the IEE method to the case study show that most of the quality goals were met. Three qualities were covered only partly by the architecture. The evaluation brought concrete benefits by revealing areas in which the existing architecture must be improved. For example, the first scenario demonstrated that two new extension points should be added. It was also demonstrated that the Model Administrator needs to be divided into two separate parts. The 3rd scenario made it evident that the product must implement a framework for building converters which translate a data model generated by one modeling tool into a format understood by another tool.

The IEE method is intended for evaluating integrability and extensibility on the architecture level (i.e. from models) and not on the implementation level (i.e. from source code). However, the modeling of the first change scenario demonstrated that sometimes code level issues may have significant impact on IE qualities. When replacing MySQL database client with variants, it was noted that all database servers may not support the SQL syntax used. This issue is undoubtedly important but cannot be seen from ordinary architectural models. Therefore, a comprehensive architectural mismatch analysis [13] is needed to verify both syntactic and semantic compliance of interfaces. Otherwise, certain implementation issues may go unnoticed when evaluating integrability and extensibility from architectural models.

The method is aimed for cost effective and repeated use by software architects. According to our experience, the scenario modeling is laborious in the beginning (i.e. before an architecture gets used to the technique), but becomes fast and easy by rehearsing. In order to make this paper accessible, the scenarios were explained in detail. In practice, an experienced architect can model scenarios rather quickly and then proceed to the evaluation phase. The scenario models may also be reused as design documents later on (i.e. if a scenario realizes there is a ready implementation plan at hand).

The case study started from a single product considered as a key product, around which the family was initiated. Thus, there is no experience on how effective the method is when it is systematically applied during the whole life cycle of a product family. In the latter case, the method may be used even hundreds of times in different situations. If the evaluation is made systematically and the evaluation results are stored in a knowledge base, e.g. in a stylebase, the collected experience can be reused in architecture development. This may lead to a situation in which the architecture meets its quality goals already after the first iteration, which means cost effective development of high quality products and more evolvable product families.

ABBREVIATIONS

AP	=	Access Point
DBMS	=	Database Management System
EMF	=	Eclipse Modeling Framework
EP	=	Extension Point
GUI	=	Graphical User Interface
HTML	=	Hypertext Markup Language
IDE	=	Integrated Development Environment
IF	=	Interface
IE	=	Integrability and Extensibility
IEE	=	Integrability and Extensibility Evaluation
MVC	=	Model-View-Controller
OS	=	Operating System
OSS	=	Open Source Software
QADA	=	Quality-Driven Architecture Design and Analysis
SQL	=	Structured Query Language
SW	=	Software
XML	=	Extensible Markup Language
UML	=	Unified Modeling Language

ACKNOWLEDGEMENTS

The publication of this paper has been supported by the Eureka ITEA research project COSI funded by the National Technology Agency (Tekes) and VTT Technical Research Centre of Finland.

REFERENCES

[1] M.G. Hardy, "COTS components in software development" in *Computer Science Discipline Seminar Conference* (CSSI 3901), 2000.

[2] C. Ncube, and M. Maiden, "COTS software selection: the need to make tradeoffs between system requirements, architectures and COTS/components" in *Workshop on Continuing Collaborations for Successful COTS Development*, 2000.

[3] A. Immonen, E. Niemelä and M. Matinlassi, "Evaluating the integrability of COTS components - the product family viewpoint" in *Commercial off the Shelf Components and Systems*, S. Beydeda and V. Gruhn, Eds. , New York: Springer Verlag, 2005.

[4] E. Niemelä. "Strategies of product family architecture development" in *The 9th International Software Product Line Conference*, 2005.

[5] Open Source Initiative, "Open source definition", http://www.opensource.org/docs/definition.php [Accessed June 2007].

[6] N. Vainio, and T. Vaden, "Sociology of Free and Open Source Software Communities: Motivations and Structures" in *Multidisciplinary Views to Open Source Software Business*, N. Helender and H. Martin-Vanhanen, Eds., Tampere: Tampere University of Technology and the University of Tampere, 2006, pp. 10-19.

[7] E. Raymond. *The Cathedral and the Bazaar*. Sepastobol: O'Reilly, 2001.

[8] J. Robbins, "OSSE practices by adopting OSSE tools", *Perspectives on Free and Open Source Software* , J. Feller, Ed., Massachusetts: MIT Press, pp. 245-254.

[9] W. Scacci, "Understanding the requirements for developing open source software systems", *IEEE Proceedings Software*, vol. 149, no 1, 2002, pp. 24-39.

[10] L. Davis, R.F. Gamble, and J. Payton, "The impact of component architectures on interoperability", *The Journal of Systems and Software*, vol. 61, 2002, pp. 31-45.

[11] A. Egyed, N. Medvidovic, and C. Gacek, "Component based perspective on software mismatch detection and resolution", *IEE Proceedings Software*, vol. 147, no 6, 2000, pp. 225-236.

[12] D. Batory., C. Johnson, B. MacDonald, and D. von Heeder, "Achieving extensibility through product lines and domain specific languages: a case study", *ACM Transaction on Software Engineering and Methodology*, vol. 11, no. 2, 2002, pp. 191-214.

[13] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it is so hard to build systems out of existing parts", in *The 17th International Conference on Software Engineering*, 1995.

[14] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture trade-off analysis method" in *The 4th IEEE International Conference on Engineering of Complex Computer Systems*, 1998.

[15] L. Dobrica, E. Niemelä ," A Survey on Software Architecture Analysis Methods", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002 pp. 638-653.

[16] E. Niemelä, M. Matinlassi, " Quality evaluation by QADA" in *5th working IEEE/IFIP Conference on Software Architecture (WICSA)* , 2005.

[17] A. Immonen,"A Method for predicting reliability and availability at architectural level" in *Software Product Lines: Research Issues in Engineering and Management*". T. Käkölä, J.C. Duenas, Eds., New York: Springer, pp. 373-422.

[18] P. Tarvainen, "Adaptability evaluation of software architectures: a case study" in *Computer Software and Applications Conference*, 2007.

[19] D. Garlan, "Software architecture: a roadmap" in *The Future of Software Engineering*, A. Finkelstein, Ed., New York: ACM Press, 2000, pp. 93 - 101.

[20] IEEE Std-1471-2000, *IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems*, 2000.

[21] K. Smolander, "Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice" in *Empirical Software Engineering International Symposium*, 2002.

[22] N. Rozanski, E. Woods. *Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. New York: Addison-Wesley Professional, 2005.

[23] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Massachusetts: Addison-Wesley, 1998

[24] B. van der Raadt, J. Soetendal, M. Perdeck, and H. van Vliet, "Polyphony in architecture" in *The 26th International Conference on Software Engineering*, 2004.

[25] P. Clements, R. Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma, "The duties, skills, and knowledge of software architects", in *The Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007.

[26] F. Bushmann., R. Meunier, H. Rohnert, P. Sammerlad, and M. Stall. *Pattern Oriented Software Architecture: A System of Patterns*. Chichester: John Wiley & Sons Ltd, 1996.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Massachusetts: Addison Wesley, 1994.

[28] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. *Second Edition.* Boston: Pearson Education Inc., 2004.

[29] R.L Glass, *Software conflict - Essays on the Art and Science of Software Engineering*, New Jersey: Prentice-Hall Inc., 1991.

[30] ISO-9126-1. Software engineering - product quality - part 1: Quality model. *ISO/IEC*, 2001.

[31] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-functional requirements in software engineering*. Boston: Kluwer Academic Publishers, 2001.

[32] L. Dobrica, and E. Niemelä, "Attribute-based product-line architecture development for embedded systems" in *The 3rd Australasian Workshop on Software and Systems Architectures*, 2000, pp. 76 – 88.

[33] K. Kronlöf, *Method integration: Concepts and case studies*. Chichester: John Wiley & Sons, 1993.

[34] E. Niemelä, and A. Immonen. "Capturing quality requirements of product family architecture", *Information and Software Technology*, vol. 49 no. 11-12, 2007, pp. 1107-1120.

[35] D. Birsan, "On plug-ins and extensible architectures", *Queue* , 2005. vol. 3 no. 2, pp.40-46.

[36] E. Clayberg, and D. Rubel. *Eclipse. Building Commercial Quality Plug-ins*. Massachusetts: Pearson Education Inc, 2006.

[37] M. Matinlassi, E. Niemelä, and L. Dobrica. *Quality-Driven Architecture Design and Quality Analysis Method. A Revolutionary Initiation Approach to a Product Line Architecture*. Espoo, Finland: VTT Publications, 2002.

[38] J. Merilinna. A *Tool for Quality Driven Architecture Model Transformation*. Espoo: VTT Publications, 2005.

[39] J. Merilinna, and E. Niemelä, "A stylebase as a tool for modelling of quality-driven software architecture" in the proceedings of the *Estonian Academy of Sciences, Special issue on Programming Languages and Software*, vol. 11 no 4, 2005.

[40] SourceForge.net, "Stylebase download statistics", http://sourceforge.net/project/showfiles.php?group_id=178714 [Accessed 15 September 2007]

[41] K. Henttonen, and M. Matinlassi, "Contributing to Eclipse: A Case Study" in *The 2007 Conference on Software Engineering* (SE2007), 2007.

[42] C. Griffin, "Tranformations in Eclipse" in *The 18th European Conference on Object-Oriented Programming*, 2004.

[43] K. Henttonen. *Stylebase for Eclipse.* An *Open Source Tool to Support the Modelling of Quality Driven Software Architecture*. VTT Research Notes 2387. Espoo: VTT Technical Research Centre of Finland, 2007.

[44] M. Fleury, and J. Lindfors, "Enabling component architectures in JVMX", *Java Online* [Online], January 2001. Available at http://www.onjava.com/pub/a/ onjava/2001/02/01/jmx.html

[45] R. Goldman, and R. Gabriel. *Innovation Happens Elsewhere. Open Source as Business Strategy*. San Francisco: Elsevier, 2005.

[46] E. Yu, "Towards modelling and reasoning support for early-phase requirements engineering" in *Third IEEE Internation Symbosium on Requirements Engineering*, 1997.

[47] E. Gamma, and K. Beck. *Contributing to Eclipse Principles, Patterns, and Plug-Ins*, Boston: Addison Wesley, 2004.

Received: August 15, 2007 Revised: October 09, 2007 Accepted: October 11, 2007