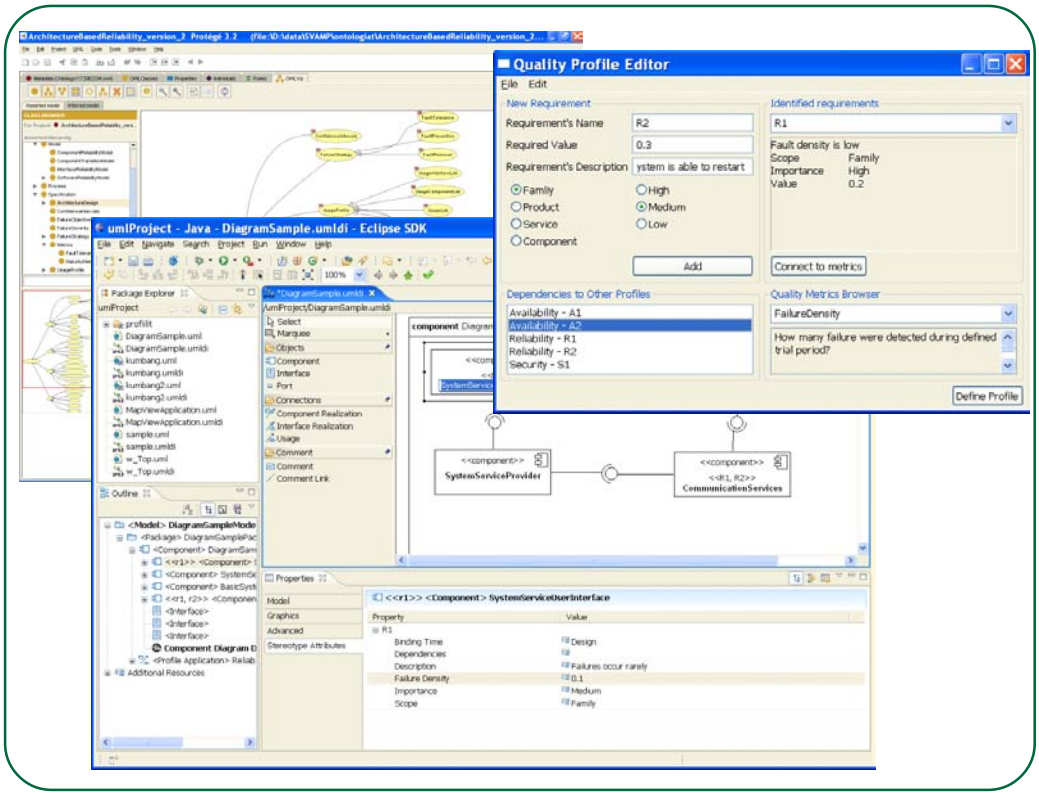
ESPOO 2007 VTT PUBLICATIONS 636



Antti Evesti

Quality-oriented software architecture development

VTT PUBLICATIONS 636

Quality-oriented software architecture development

Antti Evesti

ISBN 978-951-38-7011-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (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 Anni Kääriäinen

Evesti, Antti. Quality-oriented software architecture development [Laatuohjattu ohjelmisto-arkkitehtuurisuunnittelu]. Espoo 2007. VTT Publications 636. 79 p.

Keywords quality-oriented software architecture, software development, quality requirements, ontologies, quality meta-data management, quality modelling, quality evaluation, Unified Modeling Language

Abstract

Producing software products of good quality requires that quality requirements are taken into account as early as possible. In theory, the first place in which quality requirements can be addressed is architectural models of software. However, in practice, the software's architecture is only used to describe the functionality of the developed software. This means that the implemented software may not fulfil its quality requirements and some parts of the implementation process might be useless. The main research problem in this work is how to define and connect quality requirements with the software architecture in such a way that the requirements can vary between software components and software family members.

An environment for defining and collecting software's quality requirements was designed and implemented in this work. The environment consists of three main parts: quality meta-data management, quality modelling and quality evaluation. The quality meta-data management provides a possibility to define each quality attribute into an ontology form. These ontologies are utilized when quality requirements are defined in order to define the requirements in a uniform way in the quality modelling phase. Quality requirements are defined according to the UML profile developed for that purpose, so that it would be possible to represent these requirements in the architectural models. Finally, the architecture's quality is evaluated using evaluation tools. The purpose was to implement the whole environment on the Eclipse platform using available open source components. The Eclipse was selected because it is a widely used open source platform, which makes it easier to distribute the software developed during this work. The Eclipse tool evaluation confirmed that TOPCASED is the best available UML tool for the Eclipse and the best Eclipse ontology tool is EODM. However, EODM does not fulfil all the desired requirements of this work and this enforced the use of the Protégé ontology tool.

The implemented environment was tested using the defined test scenarios. The tests proved that the implemented environment works as expected. In addition, the developed quality profile offered an appropriate way to connect the defined quality requirements to the architectural models. There are still many things that need additional research and development before the environment and the quality profile can be utilized in software design in industry.

Evesti, Antti. Quality-oriented software architecture development [Laatuohjattu ohjelmisto-arkkitehtuurisuunnittelu]. Espoo 2007. VTT Publications 636. 79 s.

Avainsanat quality-oriented software architecture, software development, quality requirements, ontologies, quality meta-data management, quality modelling, quality evaluation, Unified Modeling Language

Tiivistelmä

Laadukkaiden ohjelmistotuotteiden valmistaminen edellyttää, että laatuvaatimukset on huomioitu mahdollisimman aikaisessa vaiheessa. Teoriassa ensimmäinen vaihe, jolloin laatuvaatimukset voidaan osoittaa, on ohjelmisto-arkkitehtuuri. Käytännössä ohjelmistoarkkitehtuurilla kuvataan suunniteltavan ohjelmiston toiminnallisuutta. Tämän vuoksi valmis ohjelmisto ei välttämättä täytä asetettuja laatuvaatimuksia, joten osa toteutuksesta voi olla käyttökelvotonta. Pääongelmana tässä työssä on se, kuinka määritellä ja liittää laatuvaatimuksia ohjelmistoarkkitehtuuriin siten, että vaatimukset voivat vaihdella ohjelmistokomponenttien ja ohjelmistotuoteperheiden välillä.

Tässä työssä suunniteltiin ja toteutettiin ympäristö ohjelmiston laatuvaatimuksien määrittelyyn ja keräämiseen. Toteutettu ympäristö koostuu kolmesta osasta: laadun määrittelystä, laadun mallintamisesta ja laadun arvioinnista. Laadun määrittelyssä jokainen laatuattribuutti määritellään ontologiamuotoon. Ontologioiden avulla laatuvaatimukset määritellään yhtenäisellä tavalla laadun mallintamisvaiheessa. Laatuvaatimukset määritellään tätä tarkoitusta varten kehitetyn UML-profiilin mukaisesti, jotta vaatimukset voidaan esittää arkkitehtuurimalleissa. Lopuksi arkkitehtuurin laatua arvioidaan arviointityökaluilla. Tarkoituksena oli toteuttaa koko ympäristö Eclipse-alustalle hyödyntäen saatavilla olevia avoimen lähdekoodin komponentteja. Eclipse valittiin, koska se on laajasti käytetty avoimen lähdekoodin alusta, mikä mahdollistaa työssä kehitettävän ohjelmiston helpon levittämisen. Toteutettu Eclipse-työkaluarviointi osoitti, että TOPCASED on paras saatavilla oleva UML-työkalu Eclipselle ja paras Eclipsen ontologiatyökalu on EODM. EODM ei kuitenkaan täyttänyt kaikkia tämän työn vaatimuksia, joten jouduttiin käyttämään Protégé-ontologiatyökalua.

Toteutettu ympäristö testattiin käyttäen määriteltyjä testitapauksia. Testit osoittivat, että toteutettu ympäristö toimii oletetulla tavalla. Lisäksi kehitetty laatuprofiili tarjosi tarkoituksenmukaisen tavan yhdistää määritetyt laatuvaatimukset arkkitehtuurimalleihin. Tarvitaan kuitenkin lisää tutkimusta ja tuotekehittelyä ennen kuin ympäristöä ja laatuprofiilia voidaan hyödyntää ohjelmistosuunnitteluun teollisuudessa.

Preface

This thesis was written at VTT Technical Research Centre of Finland, in the Software Architectures and Platforms knowledge centre. The work was done as part of the SVAMP project (Software Variability Modelling Paradigm), which is a collaboration project between VTT and HUT (Helsinki University of Technology). The project is being funded by Tekes – the Finnish Funding Agency for Technology and Innovation and VTT.

I would like to thank Research Professor Eila Niemelä for her reviews and support during this work. In addition, I give thanks to Professors Jukka Riekki and Tapio Seppänen for instructions and reviewing this work.

Oulu, February 27, 2007

Antti Evesti

Contents

Abbreviations

API	Application Programming Interface, interface to an existing application
CBSP	Component-Bus-System-Property, approach to reconcile software requirements and architecture
CVS	Concurrent Versions System, open source version controlling system
EMF	Eclipse Modeling Framework, an Eclipse plug-in for building tools and applications based on a structured data model
EODM	EMF Ontology Definition Metamodel, ontology tool for Eclipse
IEEE	The Institute of Electrical and Electronics Engineering, an international non-profit professional organization for the advancement of technology related to electricity
IEE	Integrability and Extensibility Evaluation, a method for evaluating the quality of the software architecture
ISO	International Organization for Standardization, the developer of international standards
JDT	Java Development Tools, an Eclipse plug-in for building software-based Java language
JFace	User interface framework, an Eclipse plug-in for building graphical user interfaces
MVC	Model View Controller, software architecture pattern
NFR	Non-Functional Requirement, software quality requirement
OCL	Object Constraint Language, one part of the UML specification
OMG	Object Management Group, an international non-profit computer industry consortium
OSGi	Open Service Gateway initiative (nowadays OSGi Alliance), corporation comprised of technology innovators and developers focusing on developing open service gateway technology

OWL	Web Ontology Language, a language for defining machine interpretable vocabularies specified by W3C
PDE	Plug-in Development Environment, an Eclipse plug-in for developing new Eclipse plug-ins
QADA	Quality-driven Architecture Design and quality Analysis, a methodology that provides a set of methods and techniques to develop high-quality software architectures
QASAR	Quality Attribute-oriented Software ARchitecture design method, a method for selecting the requirements of software and defining those requirements in the software architecture
QO-AE	Quality-Oriented Architecting Environment, an environment developed during this work
RAP	Reliability and Availability Prediction, a method for evaluating the quality of the software architecture
RDF	Resource Description Language, W3C specification for a metadata model
RDFS	RDF Schema, a language to structure RDF resources
SDK	Software Development Kit, a set of software development tools
SWeDE	Semantic Web Development Environment, ontology tool for Eclipse
SWT	Standard Widget Toolkit, an Eclipse plug-in for building graphical user interfaces
UI	User Interface, interface for the user to interact with a computing system
UML	Unified Modeling Language, object-based modelling technology specified by OMG
URI	Uniform Resource Identifier, unique identifier for resources
W3C	World Wide Web Consortium, standardization organization for web technologies
XML	Extensible Mark-up Language, information representation technology

1. Introduction

The software architecture design process usually concentrates on the functional requirements of the developed software. Therefore, software architectures do not contain information on the quality requirements of the software to be developed. Dropping quality requirements out of the software architecture design process may mean that a large amount of resources has been put into building a system that does not meet its quality requirements [1]. This wastes time and money and produces an architecture of poor quality.

The research problem in this work is how to connect and represent variable quality requirements in the software architecture and software components. The architecture is the first place in which software quality requirements can be addressed [2], which is why this work focuses on the tools that make it possible to connect quality requirements to the architectural models.

Quality is not an exact concept because everyone looks at quality from their own viewpoint [3]. Thus it is necessary to represent quality in a uniform way. Ontologies are utilized for this purpose in this work. The goal is that every quality attribute, e.g. reliability and security, is represented in its own ontology. However, these ontologies are not defined in this work.

Unified Modeling Language (UML) is used to represent the software architecture in this work. UML is a de facto standard for depicting software structures. Added to that, it offers mechanisms to extend its metamodel using UML profiles [4]. These profiles are a natural way to connect the defined quality requirements to the architectural models.

The purpose of this work is to design and implement an environment that makes it possible to define quality requirements of software and connect these requirements to architectural models. The primary requirements for the environment are, it should be based on a standard or widely accepted technology and it should be easy to share with architects from different companies. That is why the Eclipse platform was selected. The Eclipse platform is an open and extensible environment, both for building software and for application frameworks upon which software can be built [5]. Moreover, there are many

open source tools and plug-ins available for the Eclipse platform. This work utilizes these tools whenever possible.

The implemented environment is named the Quality-Oriented Architecting Environment, with the acronym QO-AE. The environment should offer the possibility to go through the whole quality aware architecture design procedure, from the quality attribute definition to the architecture design and finally to the quality evaluation of the designed architecture. Each of these phases needs a tool. Thus, the idea is to find a suitable tool for each phase and extend these tools in an appropriate manner. As mentioned earlier, all of these tools should run under the Eclipse platform and be available as an open source licence.

After the introduction, this thesis is structured as follows: Chapter 2 presents related research and technologies, as well as the performed tool evaluations. The environment to be developed is presented in Chapter 3. Chapter 4 discusses the implementation and test process of the environment. Finally, the results of thesis are analyzed in Chapter 5, and the work ends with the conclusions in Chapter 6.

2. Related research and technologies

This chapter first describes quality on a general level. Second, quality attributes are illustrated using different quality models. Thereafter, ontology and ontology languages are investigated. In addition, there is a brief introduction to UML 2.0. The last part of this chapter describes the Eclipse platform and available Eclipse plug-ins for UML 2.0 modelling, ontology modelling and quality evaluation.

2.1 Quality

Quality is not an exact concept because everyone looks at quality from their own viewpoint. Evans and Lindsay define quality from five perspectives [3]:

- Q1 is judgmental criteria. This defines quality as the goodness of a product. This definition is referred to as a prevalent definition of quality. The definition means that quality is absolute and universal. This kind of quality cannot be defined exactly; you just know it when you see it. This kind definition has little practical value for managers; it does not provide a way to measure or assess quality.
- Q2 is product-based criteria. This defines quality as a function of a specific, measurable variable that differs in quantity with some product attributes.
- Q3 is user-based criteria. This defines fitness for intended use, or how well the product performs its intended function. This quality definition is based on the presumption that quality is determined by what a customer wants.
- Q4 is value-based criteria. As the name says, it is based on value – that is, the relationship between usefulness and satisfaction with price. This means that the product is of good quality when it offers the same usefulness as the competitor's product, but at a lower price, or it offers better usefulness at the same price.
- Q5 is manufacturing-based criteria. This defines quality as a desirable outcome of design and manufacturing practice or conformance to the specification.

The International Organization for Standardization (ISO) defines quality in the ISO 9000 definition [6]. This definition says that quality is a characteristic that a product or service must have. For example, a product must be reliable and a service must be efficient. However, not all qualities are equal, some are more important than others. The qualities the user wants are of the most importance; these are qualities the product and service must have. Therefore, the quality of a product or service is the one that meets the wants and expectations of the user.

2.1.1 Quality attributes

The standard of the Institute of Electrical and Electronics Engineering (IEEE) [7] defines software quality as the degree to which software possesses a desired combination of quality attributes. The standard defines a quality attribute as a characteristic of software, or a generic term applied to quality factors, quality subfactors, or metric values. The terms quality attribute and quality characteristic are interchangeable [2].

Chung et al. have defined the i* framework and the non-functional requirements (NFR) framework [8]. The i* framework that is introduced first facilitates detecting where the quality requirements originate and what kind of negotiations should be taking place. The NFR framework refines the i* framework. In the NFR framework, quality requirements are called soft-goals. These soft-goals are got from the needs of stakeholders. Soft-goals help a developer to refine the quality requirements, to consider different design alternatives, to perform trade-off analyses and to evaluate the degree to which the requirements are met.

Component-Bus-System-Property (CBSP) [9] is an approach that aims at reconciling the software requirements and the architecture. It is a five-part iterative process for defining the initial architecture from the requirements. The fundamental idea of the CBSP is that every software requirement may contain information relevant to the software system's architecture. Thus each requirement is assessed for its relevance to the system architecture's components, connectors (buses), the topology of the system or a particular subsystem, and their properties. The CBSP provides an intermediate model reducing the semantic gap between high-level requirements and architectural description.

ISO/IEC standard 9126-1's quality model [10] defines quality with six categories of quality characteristics. These categories are functionality, reliability, usability, efficiency, maintainability and portability. Each of these characteristics is divided into sub-characteristics, as shown in Table 1.

Table 1. The ISO 9126-1 quality model.

Attribute	**Sub-characteristic**
Functionality	Accuracy, suitability, interoperability, compliance and security.
Reliability	Maturity, fault tolerance and recoverability.
Usability	Understandability, learnability and operability.
Efficiency	Time behaviour, resource and utilization.
Maintainability	Analysability, changeability, stability and testability.
Portability	Adaptability, installability, conformance and replaceability.

Further, ISO/IEC's technical reports 9126-2 [11] and 9126-3 [12] define intended external and internal quality metrics for measuring these quality characteristics. Internal metrics measure the software itself and external metrics measure the behaviour of the computer-based system that includes the software. These ISO/IEC reports define the metrics, the purpose of the metrics, the measurement formulas, interpretation of the measured values, etc.

Quality attributes may also be divided into two categories; execution and evolution quality attributes. Execution qualities are observable during the run time as the behaviour of the system. In other words, they constitute functional quality. Execution quality attributes are depicted in Table 2. [13]

Table 2. Execution quality attributes.

Attribute	Description
Performance	Responsiveness of the system, which means the time required to respond to stimuli (events) or the number of events processed in some interval of the time.
Security	The system's ability to resist unauthorized attempts at usage and denial of service while still providing its service to legitimate users.
Availability	Availability measures the proportion of time the system is up and running.
Usability	The system's learnability, efficiency, memorability, error avoidance, error handling and satisfaction concerning users' actions.
Scalability	The ease with which a system or component can be modified to fit a problem area.
Reliability	The ability of the system or component to keep operating over the time or to perform its required functions under stated conditions for a specific period of time.
Interoperability	The ability of a group of parts to exchange information and use the information exchanged.
Adaptability	The ability of software to adapt its functionality according to the current environment or user.

Evolution qualities are observable during a system's life cycle; they stay in the static structures of the system. In other words, they constitute non-functional qualities. Evolution attributes are depicted in Table 3. [13]

Table 3. Evolution quality attributes.

Attribute	Description
Maintainability	The ease with which a software system or component can be modified or adapts to a changed environment.
Flexibility	The ease with which a software system or component can be modified for use in applications or an environment other than those for which it was specifically designed.
Modifiability	The ability to make changes quickly and cost-effectively.
Extensibility	The system's ability to acquire new components.
Portability	The ability of the system to run under different computing systems: hardware, software or combination of the two.
Reusability	The system's structure or some of its components can be reused in future applications.
Integrability	The ability to make the separately developed components of the system work correctly together.
Testability	The ease with which software can be made to demonstrate its faults.

2.1.2 Quality-driven Architecture Design

Software architecture is described as a structure or structures of the system. Structures consist of the software components and their externally visible properties, and the relationships between them. [2]

Quality-driven Architecture Design and quality Analysis (QADA®[1]) uses quality requirements as a driving force when selecting software structures. It contributes to software family engineering by providing a method to select an appropriate family architecture approach, a method to capture and map requirements to the family architecture, a method to evaluate the maturity and quality of the family architecture, and a technique to represent variation points in the family architecture. QADA describes architecture on two abstraction levels: conceptual and concrete. Both levels are divided into four viewpoints: structural, behavioural, deployment and development. [14][15]

The QADA methodology contains several quality evaluation methods, e.g. Integrability and Extensibility Evaluation (IEE) [15] and Reliability and Availability Prediction (RAP) [16]. The IEE method includes three phases: quality goals and criteria definition, modelling system family architecture for integrability and extensibility evaluation, and evaluating integrability and extensibility from architectural models. The RAP method assists in evaluating software reliability and availability from the architectural models. The RAP method contains three similar phases: a quality requirements definition, representing quality requirements in architectural models and evaluating quality against defined quality criteria.

The Quality Attribute-oriented Software ARchitecture design method (QASAR) [1] consists of two iterative processes. The inner iteration contains three parts: functionality-based architecture design, assessment and transformation to quality requirements. The outer iteration refers to a requirement selection process. In this process some subset of requirements is selected and this subset is used within the inner iteration. As can be seen in this method, quality requirements are not the driving force in architecture development they are in the QADA method.

[1] ® Registered trademark of VTT Technical Research Centre of Finland, http://virtual.vtt.fi/qada/.

2.2 Ontology

Ontology is an explicit specification of a conceptualization [17]. Ontology is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related [22]. Ontology is a way to define the terms used to describe and represent an area of knowledge. People, databases and applications use ontologies when domain information is required to be shared. A domain means a specific subject area or area of knowledge, like medicine or food. Ontologies can define concepts and the relationships between them in a way that computers can use them. [18] There are some languages that should be used for defining ontologies in a computer readable format. These languages are presented in the next sections.

2.2.1 Resource Description Framework

The World Wide Web Consortium (W3C) defines the Resource Description Framework (RDF), a language for representing information in the World Wide Web. It is intended for representing web resources' metadata, like an author or a title of a web page. RDF also makes it possible to represent information on resources that can be identified on the Web, like items available from an online store. RDF is not intended for humans; instead, it is intended for computers and applications that process and exchange information between each other. [19]

An abstract syntax of RDF is a collection of triples, and this collection is called the RDF graph. These RDF triples are also called statements. The RDF triple consists of three parts: a subject, a predicate and an object. The subject is the identified thing, the predicate is also called the property of the triple, and the object is the value of a property. RDF uses a Uniform Resource Identifier (URI) for identifying the components of the triple. One way to represent RDF in a computer processable way is by using RDF/XML, which is an eXtensible Markup Language (XML)-based representation of RDF. [19]

For example, in the statement "The author of http://www.w3schools.com/RDF is Jan Egil Refsnes", the subject is http://www.w3schools.com/RDF, the predicate is author and the object is Jan Egil Refsnes. Using the RDF/XML syntax, this statement looks as in Figure 1. [21]

```
<?xml version="1.0"?>

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:si="http://www.secshop.fake/siteinfo#">
          <rdf:Description rdf:about="http://www.w3schools.com/RDF">
                    <si:author>Jan Egil Refsnes</si:author>
          </rdf:Description>
</rdf:RDF>
```

Figure 1. Sample statement in the RDF/XML format.

Using the RDF Validation Service of the W3C [20], this statement is represented in a graph format in Figure 2. The subject is represented by the ellipse, the arrow represents the predicate and the rectangle depicts the object.

Figure 2. Sample statement in the RDF graph.

RDF Schema (RDFS) is an extension to define application-specific classes and properties in the RDF representation. It does not include application-specific classes, but it enables the framework to define these kinds of classes. The classes in RDFS are similar to the classes in object-oriented programming languages. Thus RDFS allows defining instances of classes and subclasses of classes. [21]

2.2.2 Web Ontology Language

Web Ontology Language (OWL) is a language defined by W3C. OWL is an extension for RDF, thus it can be described using XML. OWL is intended to provide a language that is used to describe the classes and relationships between these classes. [22]

OWL is not only one language. In fact, it contains three different specifications: OWL Lite, OWL DL and OWL Full. All these languages can be derived from each other as OWL Lite is a subset of OWL DL and OWL DL is a subset of OWL Full. [24]

OWL Lite is the most simplified OWL language; it provides a classification hierarchy and simple constraint features. OWL Lite contains cardinality constraints, but it only allows the use of cardinality values of zero or one. Tool support for OWL Lite is easier to implement, unlike its more sophisticated versions. [22]

OWL DL offers maximum expressiveness without losing the computational completeness and decidability of a reasoning system. Computational completeness means that all entailments are guaranteed to be computed and decidability means that all computations are finished in finite time. OWL DL includes all OWL language constructs, but with some restrictions – such as type separation, which says that a class cannot also be an individual or property and a property for one cannot also be an individual or class. [22]

Figure 3 depicts some sample ontology that is using OWL DL [23]. This ontology defines the terms 'Pizza', 'PizzaBase' and 'PizzaTopping'. Each pizza has a base and the type of base is a PizzaBase. In addition, the ontology defines disjoints; when the class type is PizzaBase, it cannot be the PizzaTopping class or Pizza class and similar disjoints to the PizzaTopping class.

```
<owl:Class rdf:about="#Pizza">
          <rdfs:subClassOf rdf:resource="&owl;Thing"/>
          <rdfs:subClassOf>
                    <owl:Restriction>
                              <owl:onProperty rdf:resource="#hasBase/>
                              <owl:someValuesFrom rdf:resource="#PizzaBase"/>
                    </owl:Restriction>
          </rdfs:subClassOf>
          <owl:disjointWith rdf:resource="#PizzaBase"/>
          <owl:disjointwith rdf:resource="#PizzaTopping/>
</owl:Class>
<owl:Class rdf:ID="PizzaBase">
          <owl:disjointWith rdr:resource="#PizzaTopping"/>
          <owl:disjointWith rdf:resource="#Pizza"/>
</owl:Class>
<owl:Class rdf:about="#PizzaTopping">
          <owl:disjointWith rdf:resource="#Pizza"/>
          <owl:disjointWith rdf:resource="#PizzaBase"/>
</owl:Class>
```

Figure 3. Sample ontology.

OWL Full offers maximum expressiveness and the syntactic freedom of RDF without computational guarantees. OWL Full allows an ontology to augment the meaning of the pre-defined (RDF or OWL) vocabulary. It is unlikely that any reasoning software will be able to support every feature of OWL Full. [22]

2.2.3 Ontology tools

Making ontologies by hand would be hard work, especially making them in a computer readable format. Thus there are tools available to facilitate this process. The ontology editor survey by Michel Denny [25][26] identified over 50 ontology tools. The survey was updated in 2004, but because this area is evolving at every turn, the survey is only used for suggestive purposes.

Appreciable ontology tools are Protégé [27], SWOOP [28] and TopBraid [29]. TopBraid is a commercial tool and is also available for the Eclipse platform. SWOOP and Protégé are open source applications. When comparing these tools it seems that Protégé is more popular than SWOOP. There are many plug-in tools available for Protégé as well, which makes it possible to extend its functionality.

Jena is a Java framework that provides a programmatic environment for RDF, RDFS and OWL. Jena is open source and has grown out of work with the HP Lab's Semantic Web Programme. [30] From the perspective of this work, the interesting features of Jena are an RDF API and an OWL API.

2.3 Unified Modeling Language 2.0

Unified Modeling Language (UML) is a specification from the Object Management Group (OMG) [31]. UML is a language for visualizing, specifying, constructing and documenting the artefacts of software systems. In addition, UML is suitable for modelling business process and data structures. The current official version of UML is 2.0. The specification is divided into four parts: superstructure, infrastructure, Object Constrain Language (OCL) and diagram interchange. The superstructure is already completed, but the other three parts are not yet finalized. [31]

The UML 2.0 infrastructure defines base classes for UML 2.0 superstructure and Meta Data Facility (MOF) 2.0. OCL allows describing pre- and post-conditions, invariant and other conditions. UML 2.0 diagram interchange extends the UML metamodel with a package for graph-oriented information, allowing models to be exchanged and then displayed as they were originally. [31] The next section

describes the UML superstructure and some essential diagrams from the perspective of this work.

2.3.1 UML superstructure

The superstructure of UML 2.0 defines 13 diagrams [4], six structure diagrams, three behaviour diagrams and four interaction diagrams [31]. These diagrams and their relationships are illustrated in Figure 4 [4].

A component diagram defines the components used in the developed software system. A component is a modular unit with well-defined interfaces that is replaceable within its environment. The component concept addresses the area of component-based development and component-based system structuring. A component is an autonomous unit within a system and has one or more provided and/or required interfaces. Thus a component's internal structure is hidden and only accessible by the provided interfaces. [4]

A composite structure diagram is a diagram that depicts the internal structure of a classifier, as well as the use of collaboration. The composite structure diagram represents run-time instances collaborating over communication links to achieve some common objectives. [4]

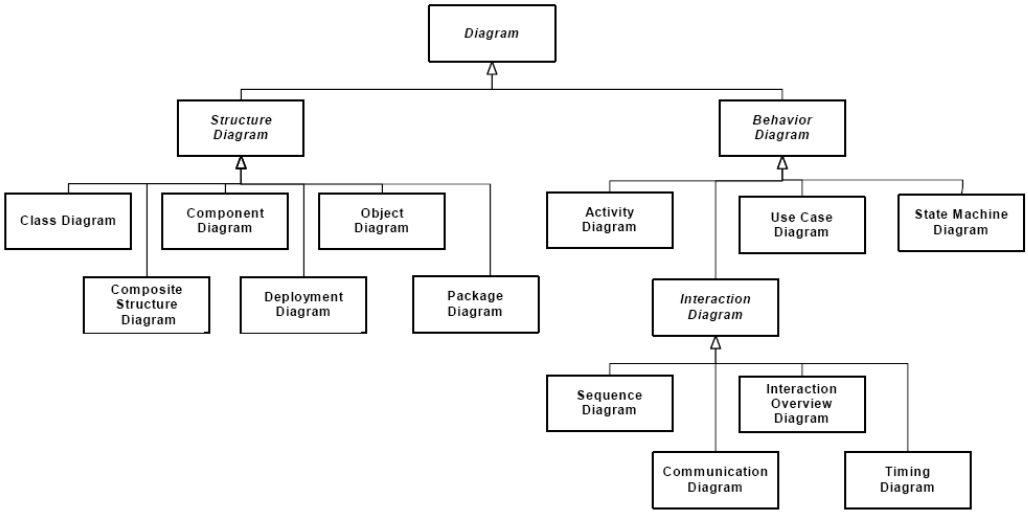


Figure 4. UML diagrams and their relationships.

A state machine diagram can be utilized to express the behaviour of a part of a system. There are concepts for modelling discrete behaviour through finite state transition systems. A sequence diagram is the most common variant of interaction diagrams. The sequence diagram depicts message interchange between participants in the interaction. [4]

2.3.2 UML profiles

UML profiles offer a way to extend the existing UML metamodel. The profile mechanism has been specifically defined to provide a lightweight extension mechanism to the UML standard. Lightweight means that the existing metamodel is not modified, only new constraints or supplemental-definitions are added to the metamodel, nothing is taken away. The intention of profiles is to give a mechanism for adapting the existing metamodel with constructs that are specific to a particular domain, platform or method. [4]

A profile includes stereotypes and constraints. When design work is based on this kind of profile, a designer has a set of concepts related to the current domain and the OCL constraints advise how these concepts should be used [32]. Metaclasses that are extended by a stereotype show up in a similar way as before, but above the name is the mark <<nameOfStereotype>> [4].

In the literature, profiles seem to be a convenient way to extend UML models, but in practice it has been seen that UML profiles are very tool-dependent. Without profile support from a UML tool, they cannot be utilized. Figure 5 presents one solution to connect stereotypes to the model element [4].

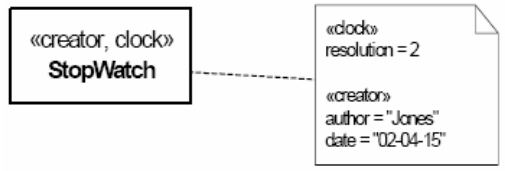


Figure 5. Stereotypes and their values.

The model element in Figure 5 contains two stereotypes: creator and clock. The content of these stereotypes is shown in a separate note in this case. Some UML

tools add the stereotype's values in the properties menu of the UML tool, thus the values are not shown in graphics.

2.4 Eclipse

IBM/OTI began developing Eclipse in 1999 and version 1.0 was published in 2001. In the same year, IBM donated the source base of Eclipse and eclipse.org was opened. Nowadays Eclipse is an open source community whose projects are managed by the Eclipse foundation. Projects under Eclipse are focusing on providing a vendor-neutral open development platform and application frameworks for building software. [33]

The Eclipse platform is an open and extensible environment, both for building software and for application frameworks upon which software can be built [5]. Usually, when speaking about Eclipse, it means the Eclipse Software Development Kit (SDK), which consists of the Eclipse platform, Java Development tools (JDT) and Plug-in Development Environment (PDE) [35] represented in Figure 6 [34]. In other words, developing Java applications can be started by means of Eclipse SDK. JDT, PDE and the Eclipse platform will be presented later on.

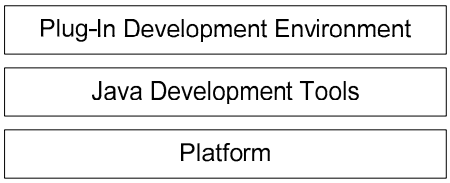


Figure 6. The three layers of Eclipse.

The smallest part of the Eclipse platform that can be developed and delivered separately is a plug-in. All the functionality the Eclipse system provides comes from plug-ins. Small tools are made up as one plug-in, but bigger tools consist of many plug-ins working together. [35] In this work, Eclipse means the whole Eclipse system – in other words, the Eclipse platform and some set of unnamed plug-ins.

JDT provides a set of plug-ins that add the capabilities of a full-featured Java Integrated Development Environment (IDE) to the Eclipse platform, supporting the development of any Java applications, including Eclipse plug-ins. JDT includes many useful features for the Java developer, such as syntax highlighting, setting break points and versatile search capabilities. [33]

PDE makes it possible to build products based on the Eclipse platform by extending the Eclipse's JDT. PDE creates a Java project that has a plug-in's nature. It offers a wizard that facilitates the creation of a skeleton of a plug-in, which might be tedious work by hand [34]. Figure 7 depicts a plug-in project's structure made by PDE.

Figure 7. Plug-in project's structure.

Through PDE, treating the content of plugin.xml, build.properties and META-INF/MANIFEST.MF files is easier because there is a graphic editor available and the user does not need to write xml by hand. These files are also called the plug-in's manifests, which contain information on the plug-in's appearance, structure and source code. The provided editor is called manifest editor. Added to this, PDE provides an easy way to run the developed plug-in, so the developer does not need to install the plug-in before making a test run. [33][34]

There are many reasons why Eclipse is selected as the used platform for this work. As mentioned earlier, Eclipse is open source software, thus everyone can download it and use it for free, which makes it easier to distribute a software developed during this work. Secondly, Eclipse offers a well-standardized environment in which to develop software. Therefore, software that runs under the Eclipse platform will work on a number of hardware platforms and operating systems.

2.4.1 Overview of the Eclipse Platform

The Eclipse Platform is a subsystem of the whole Eclipse system. It is built on top of a runtime engine. Figure 8 [35] depicts the Eclipse Platform architecture and its main components. Each of these components is implemented by using design patterns, which makes them well behaved and easy to use. For example, several components use a composite design pattern [34].

The base of the Eclipse Platform is a runtime component. It is the only part of Eclipse that is not a plug-in. The runtime component defines the plug-in model and extension points; it dynamically searches plug-ins and sustains their information and extension points. [36] The extension point is a place where things can plug into the Eclipse; likewise, extensions are things that are plugged into the Eclipse [34].

The runtime component is implemented using the Open Services Gateway initiative (OSGi) framework, especially the OSGi service model. Using the OSGi framework saves memory because the installed plug-ins are not loaded before the plug-in is needed [36]. In addition, OSGi makes it possible to restart Eclipse in the same state as it was shut down. Furthermore, the configuration can be changed while Eclipse is not running [37].

A workspace is a certain kind of resource management plug-in. It offers API for creating and managing projects, files and folders produced by other plug-ins or standalone programs and stored in the file system. [35][36] In Eclipse, the workspace is mapped directly to the file system, so there is no intermediate repository between Eclipse and the file system. This offers the user a possibility to change a resource on the file system directly or under Eclipse. [34]

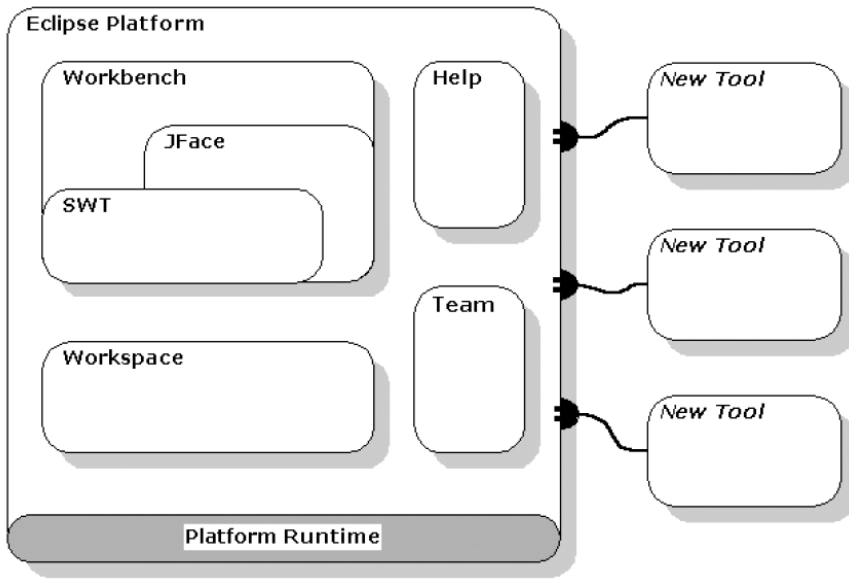


Figure 8. Eclipse Platform architecture.

A workbench is synonymous with the Eclipse Platform User Interface (UI), which a user sees when the platform is running. The workbench defines extension points for adding user interface components like menus and buttons. The workbench consists of Standard Widget Toolkit (SWT) and user interface framework (JFace), which are used to build UIs. Although Swing is a widely utilized library when developing graphical user interfaces in Java, it is not used in Eclipse. [35][36]

SWT's homepage [38] defines SWT as follows: The SWT component is designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. This is implemented by adding a thin layer on top of the operating system's native widgets. SWT provides a common set of widgets to a Java developer as buttons, menus, trees and tables, added to that it provides a layout and an event handling functionality. SWT is extended by JFace, which offers higher-level application support, like filtering and sorting tables. [34][35][38]

The Help plug-in allows tools to provide documentation in an online book format, like API documentation and user guide. The Team plug-in allows the use of versioning tools like Concurrent Versions System (CVS). [35]

In Figure 8, these three 'New tool' rectangles depict the extra functionality that is added to Eclipse by using plug-ins.

2.4.2 UML tools for Eclipse

The Eclipse project has a UML2 sub-project. The project provides the metamodel of UML 2.x for the Eclipse Platform. The UML2 metamodel is based on Eclipse Modeling Framework (EMF) implementation and is only a metamodel, not a graphical representation for UML diagrams. Figure 9 depicts a simple component diagram made using the Eclipse UML2 metamodel.

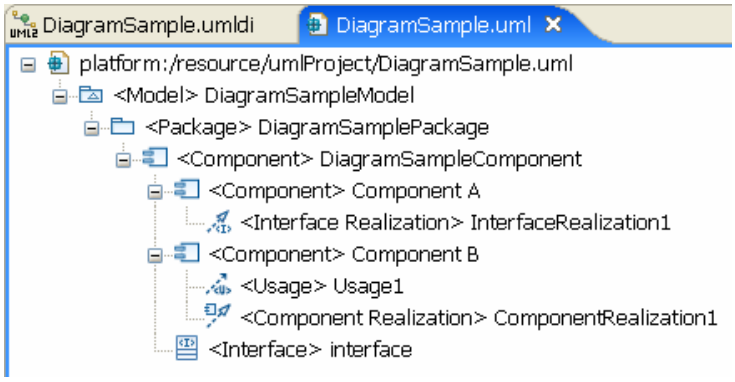


Figure 9. Eclipse UML2 model.

The model in Figure 9 contains two components, Component A and Component B, and an interface between them. The interface is used in such way that Component B should call Component A. Although the model is represented using a tree structure, using the metamodel is not a user-friendly way to modify UML diagrams. Thus Eclipse's UML2 project cannot be utilized directly. However, there are many UML2 tools for Eclipse that also contain a graphical representation of UML diagrams. Some of these tools piggyback Eclipse's UML2 metamodel, whereas others make everything in their own way.

Before making a survey of UML2 tools for Eclipse, the following base criteria were set for the charted UML tools:

- open source,
- open API or XML-based diagrams,
- possibility to exchange diagrams,
- profiles, and
- composite diagram elements.

Most of the evaluated UML tools were found from the web page of Eclipse-Plugins.info [39] in June 2006. The most promising tools were re-evaluated in October 2006. The tool supply increased during these four months. For example, the TOPCASED UML tool was published. In addition, new features were added to the existing tools, like support for the new Eclipse version 3.2, and an installation operation was facilitated. Table 4 contains the results of the tool survey. Because the majority of the tools only contain a class diagram, they were discarded and are not shown in the summary. In addition, some commercial tools were discarded due to their high price.

The survey proved that most of the tools are only toys, offering only the class diagram or storing diagrams using coordinates. When looking at the criteria and the results of the survey, it can be noticed that Poseidon does not fulfil any of these criteria and Magic Draw only fulfils one. UMLet does not use the UML metamodel, it only store diagrams in coordinates, which is an extraordinary way and makes import and export operations very difficult.

Therefore, the best UML tools available from these criteria are TOPCASED and Omondo. The remarkable thing is that TOPCASED was not available last June when the tool survey was made the first time.

Table 4. Summary of the UML2 tools for Eclipse.

Tool	Pros	Cons
Omondo EclipseUML	Stores diagrams in the XML format. Composite	A commercial tool (a free version is available). No open API in the free version. No profiles in the free version. Exchanging diagrams is prevented in the free version.
UMLet	Open source. Easy to use.	Does not make any checks on the model. Stores diagrams using coordinates. Small project.
TOPCASED	Open source. Uses the UML2 project in Eclipse. Support for UML profiles. Composite	Incomplete, version number 1.0 will be ready in the middle of 2007.
Poseidon	None.	Commercial tool. Eclipse integration is only available in the professional version.
Magic Draw	Support for UML profiles.	Commercial tool (a free version is available). No open API. Not a pure Eclipse tool; integration is difficult.

However, UML2 support for Eclipse is high-spirited and is growing all the time. Thus the results of this survey might only be valid for a few months. Both commercial and open source developers integrate their UML tools into Eclipse – apparently, software developers believe in the potential of Eclipse. This can be noticed when looking at the features of large UML tool vendors like Borland and Telelogic; they both offer Eclipse integration in their UML tools, but are beyond the scope in this work due to costs.

2.4.3 Ontology tools for Eclipse

As said in Chapter 2.2.3, there are many ontology tools available. Most of these tools are standalone programs, but this work required an ontology tool working under Eclipse, which reduced the number of tools available. An ontology tool survey was needed because there was no information on Eclipse's ontology tools

available and because existing tool survey [26] was not applicable to the scope of this work.

Before surveying the ontology tools for Eclipse, the following base criteria were set for the charted ontology tools:

- open source, and
- stores ontologies in the OWL or RDF format.

The survey of Eclipse ontology tools was done in September 2006. The survey was realized using the Internet and news groups related to Eclipse. This survey proved that there are only a few projects trying to develop an ontology tool for Eclipse, in contrast to UML2 projects.

Table 5 presents the results of this survey. The X mark denotes that the tool has a desired feature. Note that IBM's toolkit includes the EMF Ontology Definition Metamodel (EODM) and tries to extend its functionality. Thus the toolkit is not an open source but does contain useful documentation when using EODM.

Table 5. Ontology tools for Eclipse.

Tool	OWL	RDF	Graphical	Documentation is available	Open source	Active project
IBM Integrated Ontology Development Toolkit (IODT)	X	X		X		
Semantic Web Development Environment (SWeDE)	X		X	X	X	
DL-workbench	X				X	
EODM	X	X			X	
TopBraid	X		X	X		X

Open source tools were installed and evaluated using the author's own experience, whereas the commercial tools were evaluated using the information available on the manufacturer's web pages. When evaluating the activity of a project, the density of updates to the software, and project's web pages and number of news groups' posts were utilized as an indicator. TopBraid is the only project that is truly active. It seems that development of the DL-workbench and

SWeDE has completely finished. IODT and EODM were both updated in 2006, but cannot be considered active projects.

Documentation means installation instructions, a user manual or some kind of tutorial. IODT and SWeDE contain tutorials that might help with getting started, but this is not enough when thinking about efficient use. The same kind of tutorial is available for TopBraid too. Immediately after the tool survey, initial documentation was made available for EODM.

In Table 5, graphical means a tree structure or a graph – in other words, a list of ontology classes does not mean a graphical representation.

Figure 10 depicts the differences between graphical representations of Protégé and EODM. As can be seen in

Figure 10, Protégé visualizes the ontology classes in the tree structure. In addition, there are extra visualizers for Protégé. Here, the OWLViz plug-in is utilised, which draws a graph structure from the ontology. EODM only shows a list of the ontology's classes, and sub-classes are shown in the separate properties view.

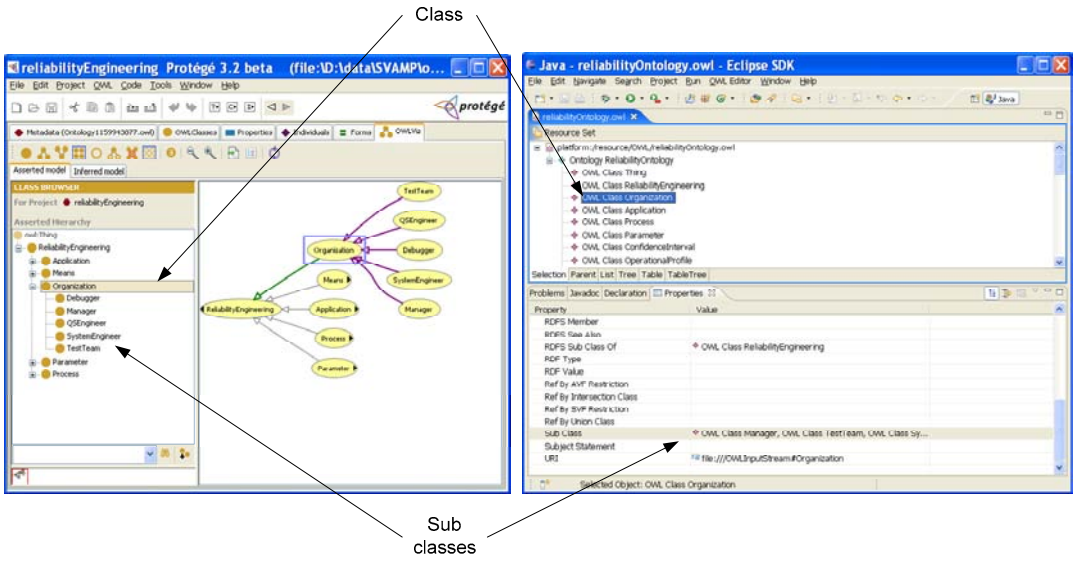


Figure 10. Protégé versus EODM.

The survey proved that open source ontology tool support for Eclipse is quite poor. The greatest weakness of these tools is project activity and lack of a graphical representation. SWeDE and TopBraid fulfil four of these hoped-for features, see Table 5, but the development of SWeDE has ended, so it is not a possible choice. Therefore, TopBraid is the best selection. However, while weighing the requirement that the tool should be open source software, EODM is the best available ontology tool for Eclipse.

2.4.4 Quality evaluation tools for Eclipse

The quality evaluation tool support for Eclipse is not as wide as the UML tool support; it is even more abridged than the ontology tool support. This can be seen when looking at the Eclipse-plugins.info web site [39]. Most of the offered quality evaluation tools are oriented towards evaluating the performance aspects. In addition, these tools always analyse a source code, not an architectural model.

While looking at the available tools it can be noticed that evaluation tools for evaluating reliability or security characteristics are not available for Eclipse at the moment. This enforces a search for quality evaluation tools that are running out of Eclipse, and then porting these tools to Eclipse.

The RAP tool [40] is this kind of standalone program. It is a tool for evaluating the software's reliability from architectural models. The tool is implemented to realize the RAP methodology [16]. In the RAP methodology, reliability and availability requirements are connected to the UML model using UML profiles and tagged values. This method makes it possible to connect the requirement's dimension and value to the model. The RAP method uses two separate profiles, the required profile and the provided profile; the first contains the desired quality and the other contains the quality the system will offer. The RAP method calculates the fault probability for each software component. Using these probabilities, a simulation model and input messages defined by the software architect the fault probability for each execution path as well as the whole system can be calculated.

The RAP tool is implemented using C# and it uses the Enterprise Architect UML tool [41]. The RAP tool seems to be a suitable evaluation tool thus it will be ported to the Eclipse platform during this work.

3. Quality-Oriented Architecting Environment

Quality-Oriented Architecting Environment (QO-AE) is aimed to facilitate the design and analysis process of quality-aware architectures. This chapter first gives an overview of the environment and then the requirements of the environment are defined. Finally, this chapter depicts the architectural design of the whole environment.

3.1 Overview

QO-AE consists of the three tools: quality meta-data management, quality modelling and quality evaluation components. These components must be integrated smoothly and co-operate perfectly, which especially means information exchanging. However, each of these components are independent and can be used separately. Figure 11 represents an overview of QO-AE.

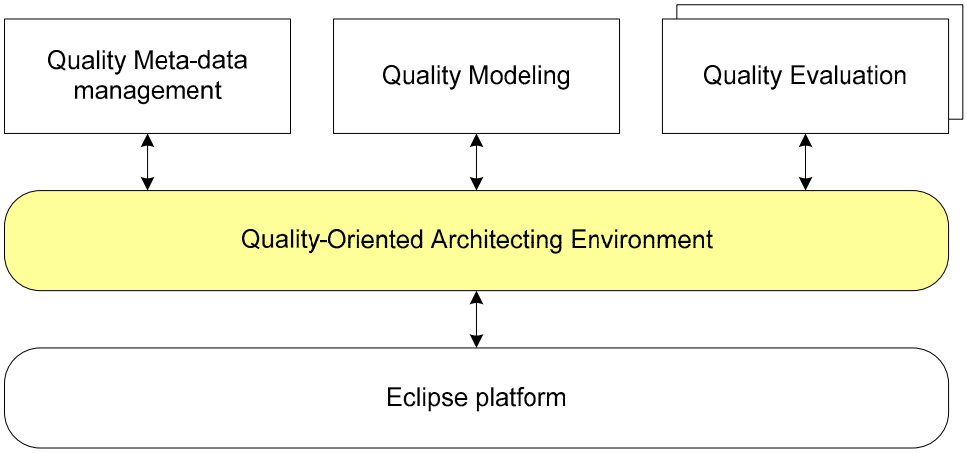


Figure 11. Quality-Oriented Architecting Environment.

The quality meta-data management component is a place where quality ontologies are defined and stored in the repository. These quality ontologies are quite stable after the defining process, but updates and enhancements are always possible. Each quality ontology defines its own quality attribute, in this case

security and reliability. The quality ontology contains metrics, unit of measurements and quantitative ranges. The metrics are the manner in which the quality level can be gauged. The unit of measurement is e.g. time, mean value or percentage. The quantitative range sets a uniform way to represent the quality values; this range is usually between zero and one. Added to this there should be a mark that shows the best value of each metric, e.g. when measuring a mean time between failures, bigger values are better, whereas when measuring a number of failures, small values are better.

The quality modelling component is a place where the quality requirements for the software to be developed are defined and mapped to the architecture. The contribution of this work is the quality profiles, which are defined using quality ontologies. Thus it is necessary to extend the quality modelling component so that the quality profiles can be constructed.

The quality evaluation component may include many evaluation tools that are used to measure the quality of the designed architecture. An example of these evaluation tools is the RAP tool. In the future, there will be tools to evaluate other quality attributes, like security and performance. The results from the evaluation tools are utilized when checking that the architecture is reaching the required quality level.

3.2 Requirements

The requirements for the QO-AE are divided into two categories: functional and quality. These requirements concern the whole environment. In addition, each separate tool has some specific requirements; these tool-specific requirements are depicted in Chapters 3.2.2, 3.2.3 and 3.2.1. Table 6 and Table 7 include the main functional and quality requirements of the QO-AE. The functional requirements mostly appear in the used techniques and desired operations. The quality requirements try to ensure that modifications and additions can easily be done in the future.

Table 6. Functional requirements for the QO-AE.

No.	Requirement	Description
F1	Eclipse plug-in	Environment is implemented and running under Eclipse
F2	Creating quality ontologies	Descriptions of the quality attributes are stored in the ontologies
F3	Reading quality ontologies	Descriptions of the quality attributes are stored in the ontologies
F4	Reading UML diagrams	Architectural models are stored using UML diagrams
F5	Storing quality-aware architecture in a uniform way	When quality requirements are added to the model, it is still formal UML
F6	Ability to create quality profiles	A software architect can select a quality ontology and create a profile based on that ontology
F7	Ability to enter quality requirements	A software architect can add quality requirements to the quality profile
F8	Requirements content is checked	The content of the entered requirements should correspond to the selected ontology

Defined requirements are numbered – the F letter in front of the number means the requirement is a functional requirement. The requirement to utilize Eclipse is the baseline because Eclipse is a widely used open source platform, which makes it easier to distribute the developed environment. Requirements F2, F3 and F4 relate directly to the utilized techniques, because quality knowledge is stored in the ontology form and architecture design in the UML form. Requirement F5 is needed so that every software architect can open and read the UML models that contain the quality information without special additions to their UML tool. F6 is required so that the UML models can be extended without a need to modify the existing UML metamodel. Requirements F7 and F8 ensure that the software architect can add quality requirements to the UML models and these added requirements are not against the rules defined in the quality ontology.

Table 7. Quality requirements of the QO-AE.

No.	Attribute	Requirement	Implementation
Q1	Extensibility	It must be possible to add new components and features to the environment	Extension points are anticipated
Q2	Integrability	Each plug-in follows Eclipse's interfaces	Eclipse's PDE carries out this requirement
Q3	Interoperability	Plug-ins are able to exchange data	Data is stored in a uniform way
Q4	Modifiability	Ability to make changes quickly	Modularity and the use of design patterns

Defined requirements are numbered, the Q letter in front of the number means that the requirement is a quality requirement. At this moment there are no performance and usability requirements for the software because the purpose is to implement the first prototype of the environment. Instead, it is important that modifications and additions can be made easily. Requirements Q1 and Q4 are directed to facilitating development work in the future. Q2 ensures that the developed environment will work with coming Eclipse versions and Eclipse plug-ins, assuming the coming versions will utilise common Eclipse standards. Requirement Q3 is needed in order to separate the parts of the QO-AE that are able to co-operate and exchange data.

3.2.1 Quality ontology tool

The ontology tool is needed to develop the quality ontologies. These quality ontologies are not made by the software architect because ontology development requires a deep understanding of the domain area. Therefore, a quality engineer defines and stores ontologies in the repository. These quality ontologies are not modified at every turn, but sometimes they need updates. Quality ontologies include a lot of knowledge, which cannot be managed by hand. Thus a program that facilitates the quality engineer's work is needed.

It is therefore necessary to select an ontology tool for use in this work. The quality meta-data management component in Figure 11 depicts an ontology tool. The requirements for the selected ontology tool are as follows:

- open source,
- Eclipse plug-in,
- represents ontologies in a graphical way,
- instructions available, and
- stores ontologies in the OWL format or offers an open API.

As this work tries to use open source tools, so the same wish is set for the ontology tool. In addition, there should be an active development group behind the tool; this ensures that software updates are available when needed. Because the purpose is to use QO-AE under the Eclipse platform, the selected ontology tool should also be an Eclipse plug-in. As an ontology tool works under Eclipse, it ensures that the ontology tool is available for each operating system and integrating the ontology tool and self-made plug-ins will be easy.

The knowledge engineer of a domain area develops the ontology. Because the knowledge engineer might not have deep expertise with ontology tools, starting to use the ontology tool should be easy and simple. Therefore, the ontology tool should represent ontologies in a graphical way, like a tree structure or a graph. Added to this, some kind of instruction is also needed.

The requirement that ontologies are stored in the OWL format or the ontology tool has to offer an open API is needed because the developed ontology will be used as an input to the quality design tool. Consequently, the selected ontology tool has to store ontologies in a uniform way, i.e. in the OWL format that can be read by other design tools. Another way to import ontologies into the quality design tool is by using an open API of an ontology tool. However, because this complicates replacing the selected ontology tool in the future, this is the last option.

When comparing these requirements with the results from the ontology tool survey in Chapter 2.4.3 and Table 5, it can be seen that there is no ontology tool that fulfils all the nominated requirements. Because the tool survey proved that ontology tool support for Eclipse is poor, the only option is to relinquish the Eclipse plug-in requirement. Several standalone ontology tools are available, as mentioned in Chapter 2.2.3. Protégé is a standalone tool that fulfils all the remaining requirements. In addition, Protégé is a broadly used ontology tool. Therefore, it is utilized for developing the quality ontologies in this work.

3.2.2 Quality design tool

The quality design tool is a tool a software architect uses to describe the software quality requirements in the architecture, i.e. the quality modelling component in Figure 11. In other words, it is a UML tool, which is extended in an appropriate way in order to construct the quality profiles. A software architect is the user of this tool.

Thus it is necessary to select a UML tool for use in this work. The requirements for the UML tool are as follows:

- open source,
- Eclipse plug-in,
- open API or XML-based diagrams,
- possibility to exchange diagrams,
- support for profiles,
- UML version 2.0, and
- composite diagram elements.

The UML tool should be open source software, because that will cut costs and offer the possibility to modify an existing code. A further requirement is that it should be an active project, so that new a version would be available when needed. The requirement that the selected tool should be an Eclipse plug-in has the same justification as the ontology tool.

Because UML diagrams are to be used as input and output for self-developed plug-ins, it is necessary that the UML tool obeys the UML metamodel and provides an open API or stores diagrams in the XML format. In addition, exchanging diagrams should be possible because different users may use the same diagrams, like the user of the modelling tool and a user of the evaluation tools.

Normally, UML diagrams only contain a representation of a system's functionality. In this case the diagrams need to be extended to contain the quality requirements as well. The most obvious extension mechanism is UML profiles, which offer a lightweight way of extending UML diagrams and components with the required quality attributes. Although the official version of UML is 2.0,

there are still many tools that use an older specification of UML. So it is necessary to add a requirement to support the specification of UML 2.0.

Added to this, the UML tool should support compositing elements in the UML diagram in order to depict software components inside another component. Composite elements are specified in the UML 2.0 specification [4], although not all UML tools that support UML 2.0 include composition characteristics.

Looking these requirements for the UML tool and the results of the UML tool survey presented in Chapter 2.4.2, only the TOPCASED UML tool offers all the required functionality and features. The Omondo EclipseUML is another alternative but it does not offer model extensions, like profiles. In addition, diagram exchanging is prevented in the free version of Omondo.

Therefore, the only possible UML tool is TOPCASED. This tool is still under construction, but the current version offers all the required functionality. TOPCASED also uses Eclipse's UML2 project, and this makes import and export operations easier because instructions are available on the Eclipse web sites. Figure 12 represents a simple model made using the TOPCASED UML tool. This model is the same as that shown in Figure 9, but in a more readable format. However, TOPCASED was not available when the first part of this work started, so Omondo is also used in some parts of this work.

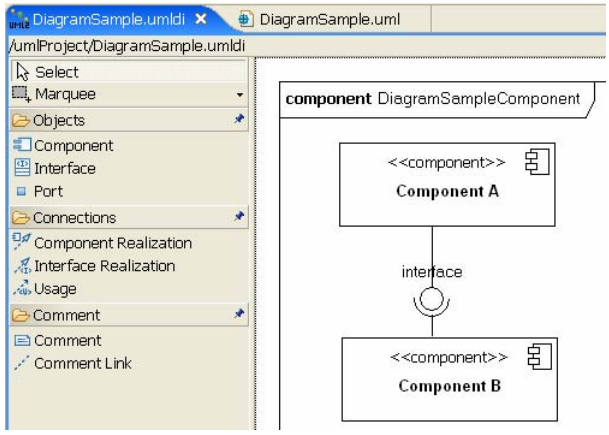


Figure 12. TOPCASED UML model.

TOPCASED stores UML diagrams by using two files per each diagram. One contains graphical information on the diagram, and the other includes diagram information in the pure Eclipse UML2 format. It is also possible to import diagrams to TOPCASED using just the latter; in that case the user can drag-and-drop all elements to a graphical editor and the program ensures that all relationships between the elements survive.

3.2.3 Quality evaluation

As mentioned in Chapter 2.4.4, there was no quality evaluation tool available for Eclipse. Without an applicable quality evaluation tool, QO-AE will only be an environment in which to collect and connect quality requirements to the UML models. Thus porting the RAP tool to Eclipse is a reasonable choice.

The functional requirements for Eclipse's RAP tool are quite similar to the former RAP tool because the purpose is to make the same evaluation possibility available to Eclipse. These requirements are as follows:

- Read the UML state diagram, sequence diagram and activity diagram.
- Calculate the fault probabilities for components, execution paths and the whole system.

The quality requirements for Eclipse's RAP tool contain the same quality requirements as for the whole QO-AE. These requirements are listed in Table 7. The following requirement is also defined:

- easy to port to different UML tools.

The first version of the RAP tool had the same quality requirement. The possibility of changing the UML tool is important because in the future there might be features the selected UML tool does not fulfil or a new UML tool that fulfils the desired requirements better than the first tool may be found.

3.3 Architecture

This section represents the architectural design of the QO-AE. The structural view of the whole environment is represented at first and thereafter the

behavioural view of the environment. After that, the structure of the quality ontology and the quality profile is represented. Readymade software is mostly utilised. However, some components require their own design and implementation. Therefore, the design of these components is also represented here, i.e. a structural view of the profile editor and the RAP tool.

3.3.1 Structure

The structure of the whole QO-AE system is depicted in Figure 13. The rectangles depict independent software components and the ellipses depict transmitted data, i.e. the content and file format. Protégé is represented outside the rounded rectangle because it is not running under Eclipse and the purpose is that QO-AE can be used under the Eclipse Platform.

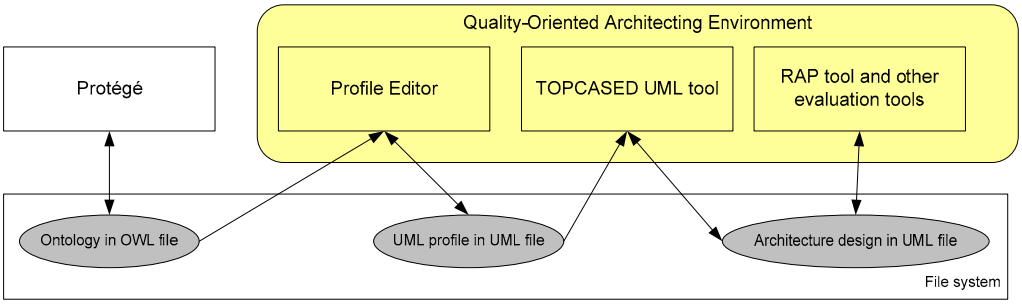


Figure 13. Phases of the Quality-Oriented Architecting Environment.

Figure 13 shows that whole process consists of four phases. When compared with Figure 11, it is remarkable that the quality design component is divided into two parts: Profile editor and UML tool. This is because it will better depict the use of the quality design component. The first phase is quality ontology definition; each quality attribute is defined in its own ontology and stored in the repository in the OWL format. The second phase is defining a quality profile using the previously defined quality ontology. In the third phase this quality profile is connected to the architectural model and, finally, in the fourth phase the architecture's quality is evaluated using the evaluation tool and the results of the evaluation process are also stored in the architectural models. A detailed description of each phase is given in Chapter 3.3.2.

The QO-AE is dependent on the exchanged data transmitted via a file system, as shown in Figure 13. Using the existing file system ensures that data can be transmitted using standard and commonly used techniques like OWL files and the UML project files of Eclipse. Because the majority of the available ontology tools store ontologies in OWL files, in the future it is possible to replace Protégé with another tool, for example EODM, without a need to change other parts of the QO-AE. The same thing applies to the UML tool; apart from TOPCASED, most UML tools for Eclipse use Eclipse's UML files. Thus this implementation technique makes it possible to change the ontology tool or UML tool in the future if necessary.

The content and format of the data is also depicted in Figure 13 using ellipses. Arrows show how the data can be moved and modified. Thus the profile editor can only read the ontology and the TOPCASED UML tool can only read a profile. They cannot exchange this data, as the one-sided arrow shows. Vice versa, a dual-headed arrow depicts that the data can be read and modified. A notable thing is that evaluation tools cannot modify the architecture design, but they can add the results of the evaluation to the architecture model.

As mentioned earlier, the Protégé ontology tool and the TOPCASE UML tool are ready to use. The parts that need to be implemented are the profile editor and the RAP tool, i.e. porting the existing RAP tool to Eclipse.

3.3.2 Behaviour

Figure 14 contains the activity diagram of the QO-AE. Each swim-lane represents one stakeholder in the system and the rounded rectangles depict an activity that should be taken in the current stage. The white boxes mean input information on an activity or output information that should be got from the activity. The system's execution starts from the black circle in the upper corner and ends at the black circle with a ring in the bottom corner.

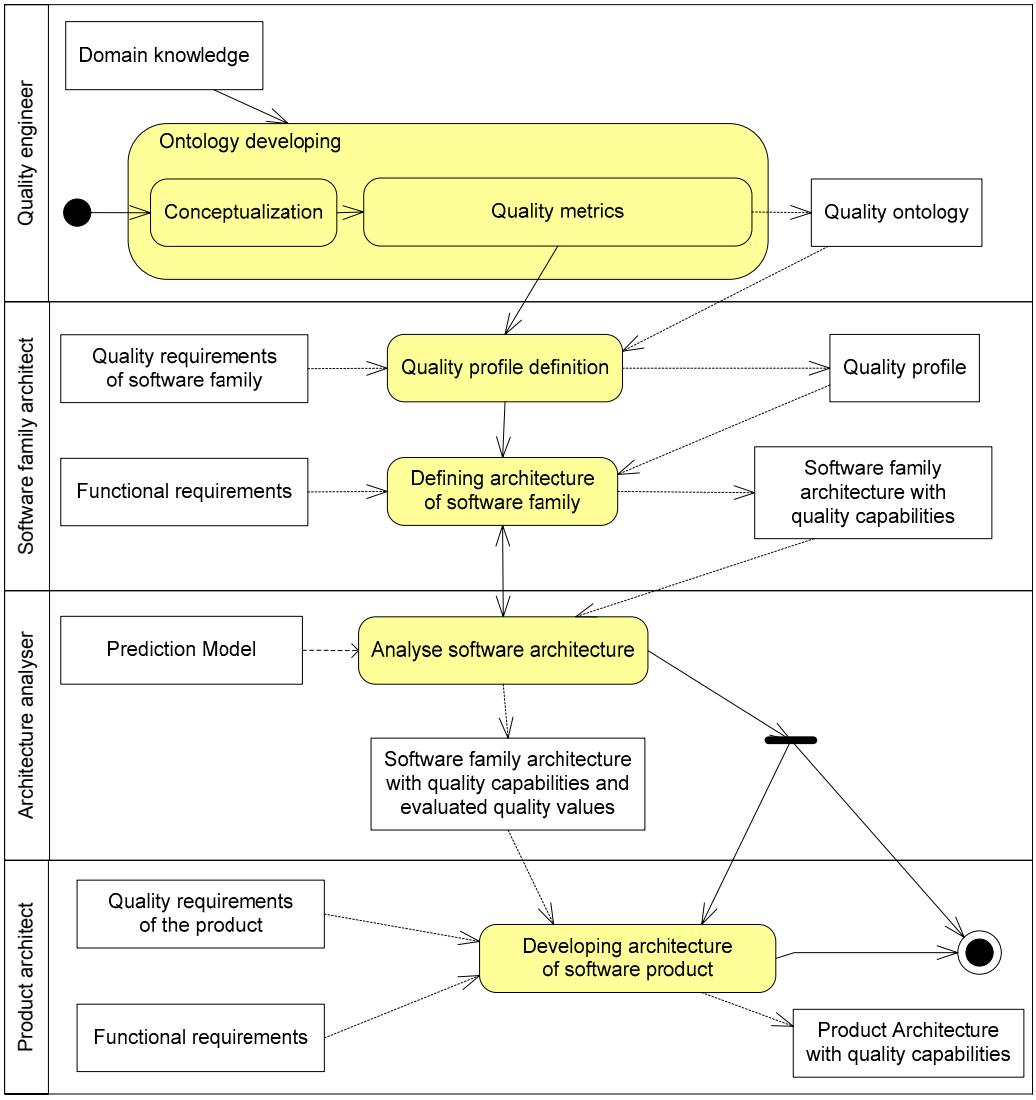


Figure 14. Activity diagram of the QO-AE.

The stakeholders in the whole process are quality engineers, software family architects, architecture analysers and product architects. The dual-headed arrow between the *Defining architecture of software family* and *Analyse software architecture* actions means that the process is iterative, so the family architecture requires enhancements if the desired quality levels are not met.

In this work the purpose is to address the tools of a quality engineer, a family architect and an architect analyser. The role of a product architect is beyond the scope of this work. Because of this there is fork-join element in the product derivation in the swim-lane of the architecture analyser. Going further from the fork-join element requires that the family architecture is analysed and the desired quality levels are met.

The action series of the quality engineer while defining a quality ontology is as follows:

1. The quality engineer defines concepts of the knowledge area.
2. The quality engineer gives units and ranges for quality measurements.
3. The quality engineer stores the defined ontology in the OWL format.

The action series of the software family architect while defining a quality profile is as follows:

1. The software family architect opens a quality ontology file.
2. The software family architect defines the quality requirements for the software family.
3. The software family architect gives an importance level to each quality requirement.
4. The software family architect gives a desired quality value to each quality requirement.
5. The software family architect maps each quality requirement to some metric from the opened ontology.
6. The software family architect selects the profiles that will affect the new profile, i.e. dependencies between profiles.
7. The software family architect gives a command to validate and store the profile.

The action series of the software family architect while designing a family architecture is as follows:

1. The software family architect designs the software family architecture.
2. The software family architect applies the previously defined quality profiles to the architecture.
3. The software family architect selects the components to which a stereotype from the profile is connected.

4. The software family architect can modify the importance level and quality values of each quality requirement, as well as other profile variables.
5. The software family architect saves the finished family architecture.

The action series of the architecture analyser is as follows:

1. The architecture analyser opens the architecture model.
2. The architecture analyser gives input information and analyses the system.

The action series of the product architect is as follows:

1. The product architect selects a family architecture.
2. The product architect makes the variants required for the product; some specific components may be added.
3. The product architect defines the product-specific quality requirements and maps them to the architecture. These requirements are some kind of enhancements to the quality requirements for the family architecture.

3.3.3 Ontologies and profiles

Figure 15 [42] depicts part of the architecture-based reliability ontology, which describes the reliability attribute. This kind of description is needed for each quality attribute.

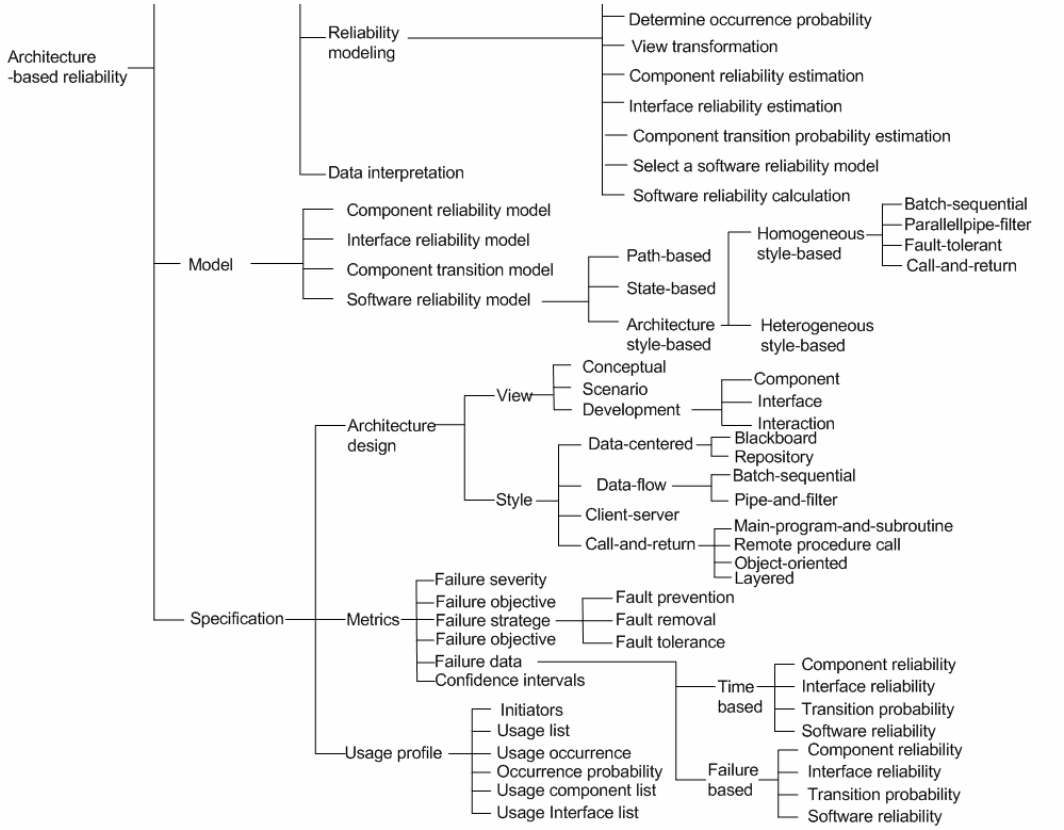


Figure 15. Part of the reliability ontology.

The ontology in Figure 15 specifies the primitive concepts of the software architecture's reliability. Most of the information is beyond the scope of this work because the purpose is to connect the quality requirements to the architectural models, thus the Model and Architecture design classes are not addressed here. But the Metrics class under the specification class is interesting. The Metrics class contains classes that can have specialized software quality metrics, such as are represented in Table 8. The reliability metrics presented in Table 8 are defined in the IEC report [11].

Table 8 represents six ways of measuring software reliability. The first column contains the name of the metric, which is also called a measurement unit in this work. The purpose of the metric is presented in the second column in a question format. The third column includes a measurement formula for the metric and an

explanation of the data elements. Added to this third column shows the range and preferred values of the metric. The range of the *Mean time between failures* and *Mean down time* is between zero and infinity. So it is necessary to normalise these values to the range zero to one in order to facilitate a comparison process.

Table 8. Reliability metrics and purposes.

Metric name	**Purpose of the metric**	**Formula and value range**
Fault density	How many faults were detected during the defined trial period?	X = A / B A = number of detected faults. B = product size X ≥ 0, 0 is the best value
Mean time between failures	How frequently does the software fail in operation?	X = T / A T = operation time. A = total number of actually detected failures. X > 0, bigger values are better.
Breakdown avoidance	How often does the software product cause the breakdown of the total production environment?	X = 1 – A / B A = number of breakdowns B = number of failures. 0 ≤ X ≤ 1, 1 is the best value
Mean down time	What is the average time the system stays unavailable when a failure occurs before gradual start-up?	X = T / N T = total down time. N = number of observed breakdowns. X > 0, bigger values are better.
Restartability	How often can the system restart providing service to users within a required time?	X = A / B A = number of restarts which met the required time during testing or user operation support. B = total number of restarts during testing or user operation support. 0 ≤ X ≤ 1, 1 is the best value
Restorability	How capable is the product in restoring itself after an abnormal event or on request?	X = A / B A = number of restoration cases successfully done. B = Number of restoration cases tested as per requirements. 0 ≤ X ≤ 1, 1 is the best value

Quality profiles are the contribution of this work. These profiles make it possible to connect quality requirements to the architectural models, i.e. UML diagrams. The quality profiles are constructed using the profile editor, a quality ontology and the quality requirements for the developed system.

The quality profile consists of stereotypes, and each stereotype describes one quality requirement. Each stereotype contains the requirement's name, description and metric, and scope, importance and dependencies field. The name field contains the requirement's identification name, e.g. R1 means the first reliability requirement. The name will also be the name of the stereotype, which is shown when the requirement is connected to some UML element in the architecture. Because the requirement name is only an abbreviation, there is a separate description field that contains a detailed description of the requirement, e.g. 'System is able to restart'. The metric field tells which metric should be used when measuring that requirement. The metric is got from the quality ontology and its value can vary within the range set in the ontology, e.g. if R1 is using the Restartability metric, it can get values between zero and one, as shown in Table 8. The rest of the quality profile's fields, i.e. scope, importance and dependencies, are got from the Quality variability ontology presented in Figure 16 [44].

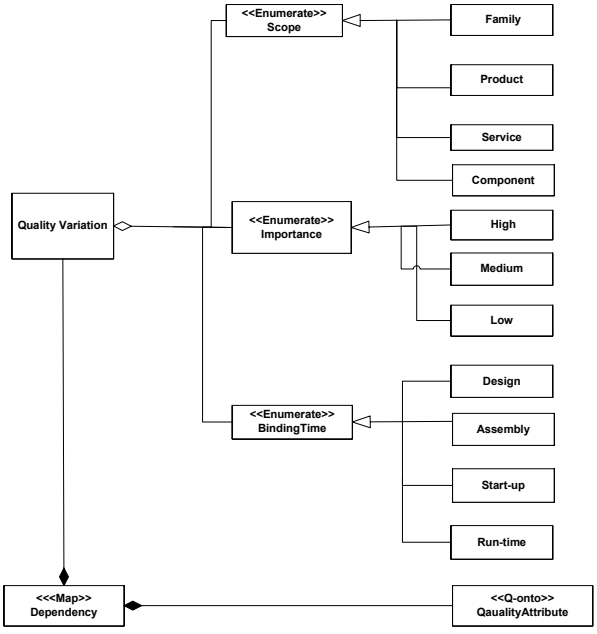


Figure 16. Quality variability ontology.

As the quality variability ontology shows in Figure 16, the Importance and Scope fields can get predetermined values. The possible values of importance are high, medium or low, and values for the Scope are family, product, service

or component. The stereotype's dependencies field describes the dependencies on other profiles, thus it contains a list of names of profiles. Every stereotype in a profile has the same content in the dependencies field, because the field contains dependencies between profiles not dependencies between stereotypes. For example, quality profiles availability and security can have dependency because increasing security may decrease the availability.

The quality profile should be represented in the format of Eclipse's UML project because the TOPCASED UML tool can exploit this format. TOPCASED and Eclipse's UML project stores diagrams in the XML format. To produce a UML profile by writing XML statements would be laborious and error prone; therefore, Eclipse's UML project offers code libraries that can be utilised when constructing profiles.

3.3.4 Structural view of the Profile editor

The purpose of the Profile editor is to construct the quality profile as described in Chapter 3.3.3. The list below represents the actions the Profile editor takes when constructing a quality profile:

1. The user opens a quality ontology.
2. The Profile editor creates a list of available quality metrics based on the opened ontology. Each item has a range and the best value of measurement.
3. The user enters all the quality requirements for the system to be developed by the Profile editor.
   a. Finally, the Profile editor contains a list of all the quality requirements.
   b. The user selects a metric for each quality requirement; these metrics are got from the ontology.
4. The user defines the dependencies between the new profile and the previously defined profiles.
5. The Profile editor validates the profile.
   a. Check that requirement's value belongs to the valid range of the selected metric.
   b. Check that the requirement's name is unique.
6. The Profile editor stores the valid profile.

There are two alternative architectural patterns that could be used when designing the architecture of the Profile editor: Blackboard and Model View Controller (MVC) [43].

The Blackboard is useful for problems for which deterministic solution strategies are not known. In Blackboard, several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution. MVC divides an interactive application into three components: model, view and controller. The model contains data and functionality. Views display information to the user. User inputs are handled by controllers. Views and controllers together comprise the user interface and a change-propagation mechanism ensures consistency between the model and the user interface. [43]

When analysing these architectural patterns it seems the MVC pattern is better suited to the Profile editor. The argument for this selection is that the function of the Profile editor is strongly dependent on user inputs and these inputs are got from the user interface, i.e. from the view. Thus MVC is selected for the Profile editor. Figure 17 represents the component diagram of the Profile editor using the MVC pattern.

In Figure 17, the controller and model components are named as the MVC pattern assumes, but the view component is named to the GUI. The first implementation of the Profile editor can be realized using one view controller pair, although MVC gives the possibility of using several view controller pairs.

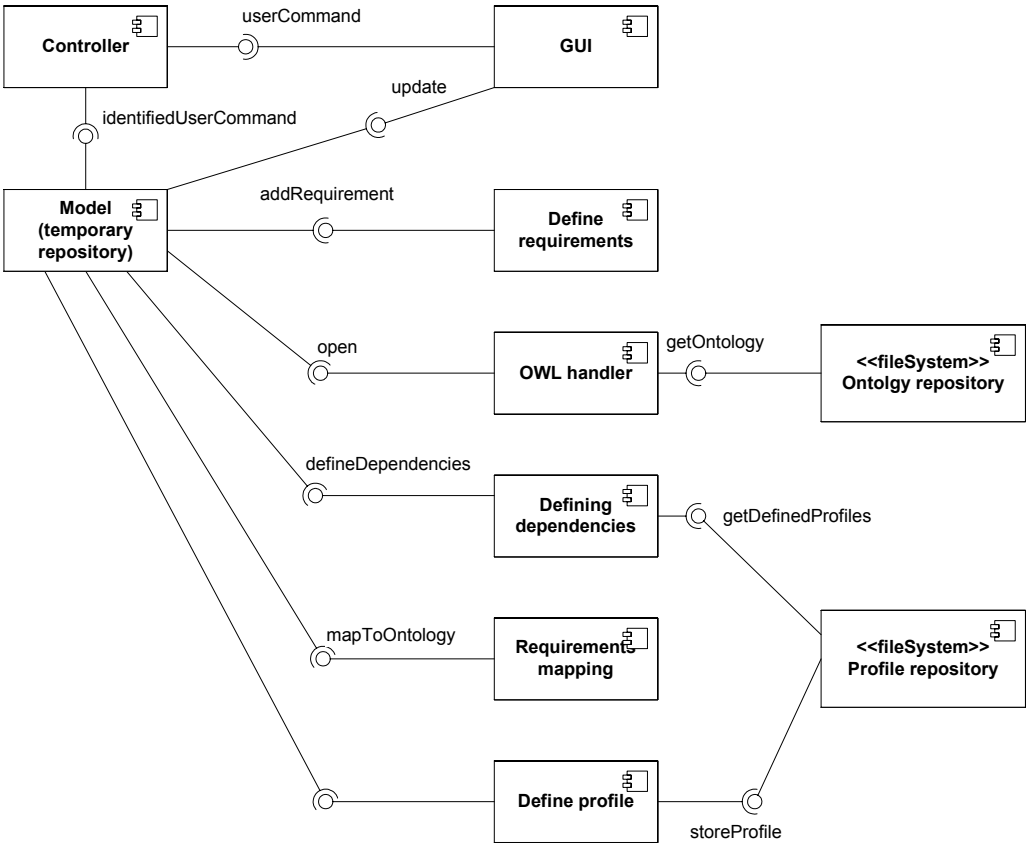


Figure 17. Component diagram of the Profile editor.

Figure 17 shows that user commands are passed to the model via the controller component. In the MVC architecture the controller component takes user inputs, and the interface between the GUI and the controller illustrates that the user uses the GUI by entering inputs. Depending on the command, the model component calls the appropriate functional component. The OWL handler component gets the ontology from the Ontology repository, which is implemented as a file system. A similar repository is used to store the quality profiles, called the Profile repository. Defining dependencies and Define profile components use that repository. Whenever the content of the model component is changed, it will call the GUI's update method in order to show the view's state.

3.3.5 Structural view of the RAP tool

The RAP tool is designed to use the Blackboard architectural pattern rather than the MVC model. The reason for this selection is that the RAP tool's operation is mostly data-centric, like reading information from the existing UML diagrams and calculating the results from this information. Thus the input from the user is insignificant. Another reason for the selection is that the first version of the RAP tool was implemented using the Blackboard architectural pattern, and that was found to be a good selection. Figure 18 depicts the architecture of the RAP tool using a component diagram.

In Figure 18 the common elements of the Blackboard pattern are depicted on the left and right side, i.e. the controller and Blackboard components. The independent execution units, which are called by the controller component, are presented in the middle of the Figure. The Blackboard component does not know anything about the other components as it only stores data; the other components inspect and update this data.

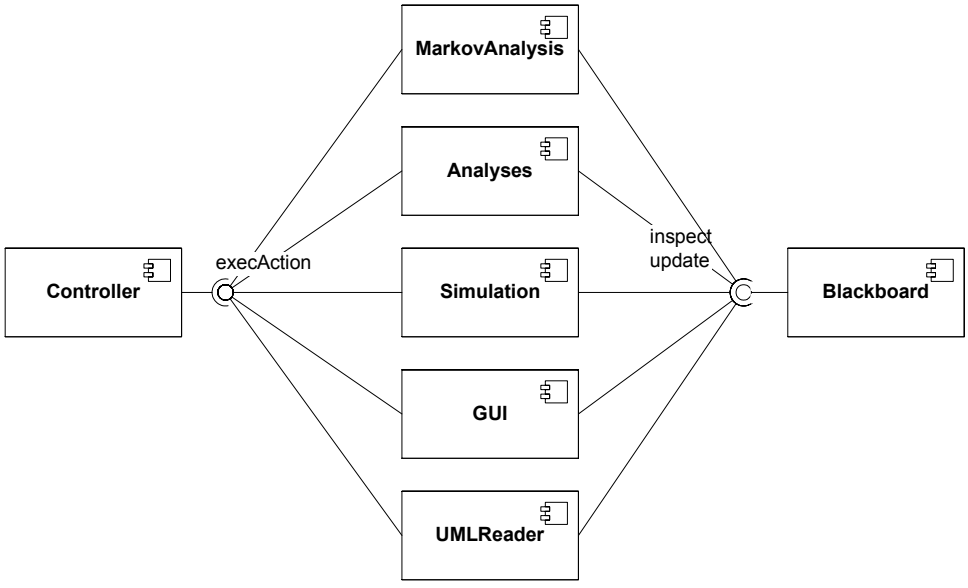


Figure 18. Component diagram of the RAP tool.

4. Implementation and testing

This chapter presents the implementation and testing of the QO-AE. The purpose is to present some of the utilised design patterns that are important for the implementation, not to describe detailed implementation like code or class diagrams. In addition, examples will clarify the whole QO-AE process.

4.1 Ontology under Protégé

The reliability ontology depicted in Figure 15 is not applicable as such because it does not contain measurable reliability metrics, as shown in Table 8. Thus it is necessary to combine the information from Figure 15 and Table 8. As stated in Chapter 3.3.3, the Metrics class is the right place to add reliability metrics. A parent class for each metric is selected by looking at the purpose of the metric and negotiating with the developer of the reliability ontology. Table 9 shows the same metrics as Table 8 and the selected parent classes.

Table 9. Reliability metrics and their parent classes.

Metric name	**Parent class**
Fault density	FailureData
Mean time between failures	TimeBased
Breakdown avoidance	FailureBased
Mean down time	TimeBased
Restartability	FaultTolerance
Restorability	FaultTolerance

This classification is not absolute and there are other possibilities when selecting parent classes. However, this solution is used as a starting point.

Each metric in the ontology has to include a value range and a mark that denotes a direction when the value is getting better. Properties are natural way to connect these kinds of things to the ontology. Connecting numerical properties to the ontology requires that the subject is defined as an instance, thus each metric is described using instances called individuals in Protégé. Each metric instance has the properties minValue and maxValue, which are used to describe the value

range of the metric. Added to this, the metric has the property targetValue, which denotes the metric's best value. Every value range is normalised between zero and one because some metrics may get values to infinity. Therefore, the selected properties are applicable to every metric.

It is also possible to add comments to the ontology elements. This possibility is utilised in entering the purpose of the metrics in the reliability ontology. Consequently, the comment on each metric contains the same description as in Table 8's column Purpose of the metric. This facilitates further development of the reliability ontology.

Figure 19 depicts the reliability ontology in Protégé. The window consists of three vertical direction areas; a *class browser* on the left side, an *instance browser* in the middle and an *individual editor* on the right side. These views give a possibility to add a new content to the ontology and edit the existing one.

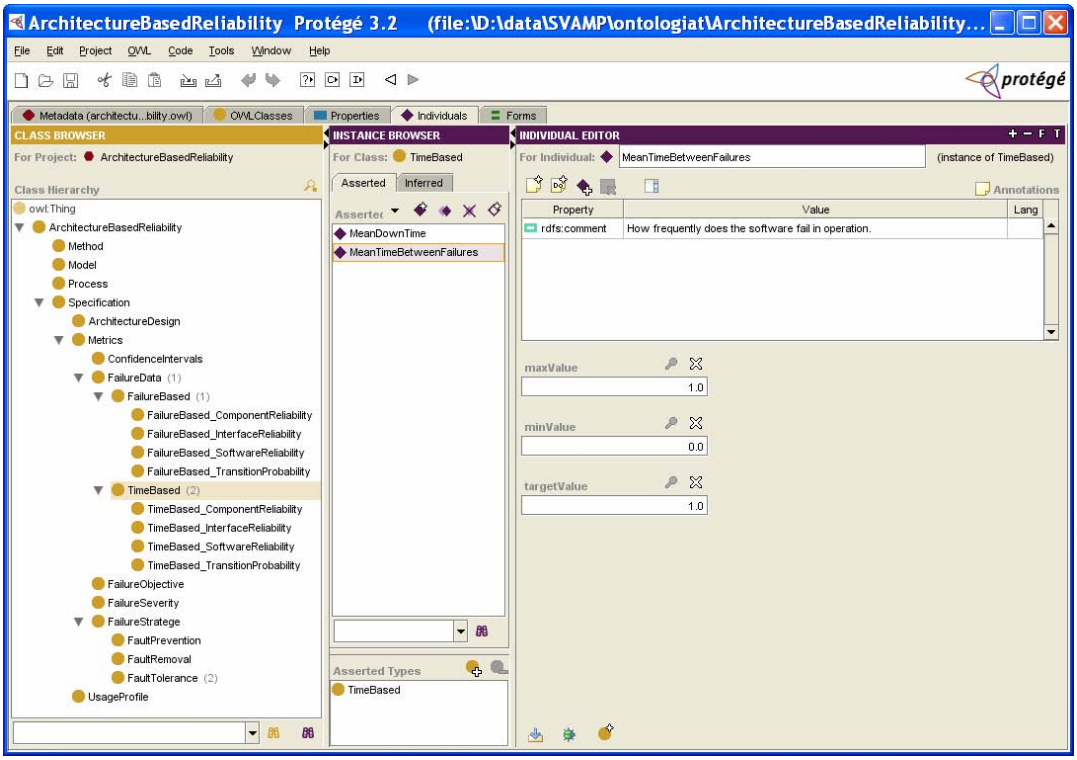


Figure 19. Reliability ontology in Protégé.

In Figure 19 the class browser contains the classes of the reliability ontology as a tree structure. The number behind the class name tells how many instances that class has, i.e. how many metrics that class has. The middle part of the Figure contains instances of the selected class, and instances represent metrics in this case. Thus the *TimeBased* class has the metrics *MeanDownTime* and *MeanTimeBetweenFailures,* as defined in Table 9. The part on the right side depicts the content of the selected metric. The content of the metric is similar to that described in Table 8, but the value range is normalised so that the minValue is zero and the maxValue is one.

When the quality engineer has defined the quality ontology in Protégé, the ontology is stored in the file system in the OWL format, as shown in Figure 13.

4.2 The Profile editor

The Profile editor works like an Eclipse plug-in, thus it is implemented using Java language and Eclipse's plug-in development environment, which was presented in Chapter 2.4. In Chapter 3.3.4 the Model View Controller was selected for the architecture of the Profile editor. The MVC uses a change-propagation mechanism to ensure that the model and views stay in a consistent state. In [43] the change-propagation mechanism is implemented using an observer design pattern. The observer design pattern implements a one-to-many dependency between participant objects. The observer is constituted of four classes: Subject, ConcreteSubject, Observer and ConcreteObserver [45]. Figure 20 represents the observer design pattern applied to the MVC architecture.

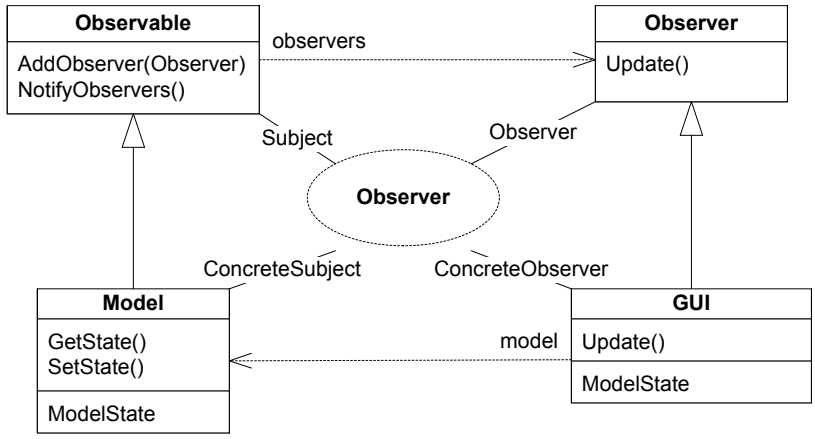


Figure 20. Observer design pattern adapted to the MVC architecture.

As Figure 20 shows, the Model extends the Observable class and the GUI extends the Observer class. The Model class corresponds to ConcreteSubject and the GUI class corresponds to ConcreteObserver. The AddObserver method offers a way to add an observer to the Model and the NotifyObservers method informs observers – i.e. the GUI class – when the Model is changed. The update method is used when the Model is changed to update the ModelState attribute in the GUI class.

Sun's Java library offers complete classes for implementing the observer pattern, i.e. Observable and Observer classes. Thus this ready-made part is utilised in the Profile editor implementation. The final GUI of the Profile editor is presented in Figure 21.

Figure 21. GUI of the Profile editor.

The Profile editor's GUI includes four main groups: *New Requirement*, *Dependencies to Other Profiles*, *Identified requirements* and *Quality Metric Browser*. Added to this, the GUI contains a menu bar and a Define Profile button.

The user can open a quality ontology by using a file menu on the menu bar. The Open ontology operation is implemented in the OWL handler component using the Jena library [30], i.e. the Profile editor reads the OWL file that Protégé stored. The content of the opened ontology is displayed in a drop-down menu of the *Quality Metric Browser* group. The edit menu offers a possibility to select a folder of quality profiles; the content of that folder is displayed in a list of the *Dependencies to Other Profiles* group. This list gives a possibility to select profiles, which will affect the new profile.

The user can add new quality requirements using the *New Requirement* group. This group collects all information on the requirement apart from the used metric. All added requirements are listed in a drop-down menu of the *Identified*

Requirements group. The user connects a defined requirement to the metric in the ontology as follows:

1. The user selects a requirement from the drop-down menu of the *Identified Requirements* group.
2. The user selects a metric from the drop-down menu of the *Quality Metric Browser* group.
3. The user presses the *Connect to metric* button.

When the selected requirement is connected to a metric, the Profile editor checks that the requirement's value is included in the range of the selected metric. The *Define Profile* button starts a profile definition using the information the user has entered. Profile creation is implemented using code libraries, which are included in Eclipse's UML project. The profile is assembled from stereotypes, as described in Chapter 2.3.2, and each stereotype represents one defined quality requirement, as mentioned in Chapter 3.3.3. First, the program asks the name of the new profile and thereafter the profile is stored in the selected profile folder, i.e. in the existing file system in the UML format, as shown in Figure 13.

4.3 Quality profile and TOPCASED

When the quality profile is defined and stored in the profile folder it can be connected to the UML model, which is stored in Eclipse's UML format. The Profile editor defines the profile's stereotypes so that the stereotypes can be connected to the component elements in the UML diagram. Figure 22 depicts a component diagram that is opened under Eclipse with the TOPCASED UML tool. Figure 22 also shows the situation after the software architect has connected the previously defined reliability profile to the components of the model.

The files of the UML design project are on the left side of Figure 22. The list also contains the profiles folder, and under the folder is the previously defined reliability profile, i.e. the profile named *ownReliability.profile* that was defined in Chapter 4.2. This profile is applied to the component diagram called DiagramSample, which is intended to show how to connect the quality requirements to the architecture model. The software architect has connected the

stereotypes from the reliability profile to elements of the diagram so that the *SystemServiceUserInterface* component contains all three stereotypes R1, R2 and R3, whereas the *SystemServiceProvider* component contains stereotypes R2 and R3 and the *CommunicationServices* component only contains stereotype R1. Each of these stereotypes represents one quality requirement, as stated in Chapter 3.3.3. Connecting quality requirements to the UML model can be done using the normal operations the UML tool offers because the quality requirements are defined using stereotypes that are supported by the TOPCASED UML tool.

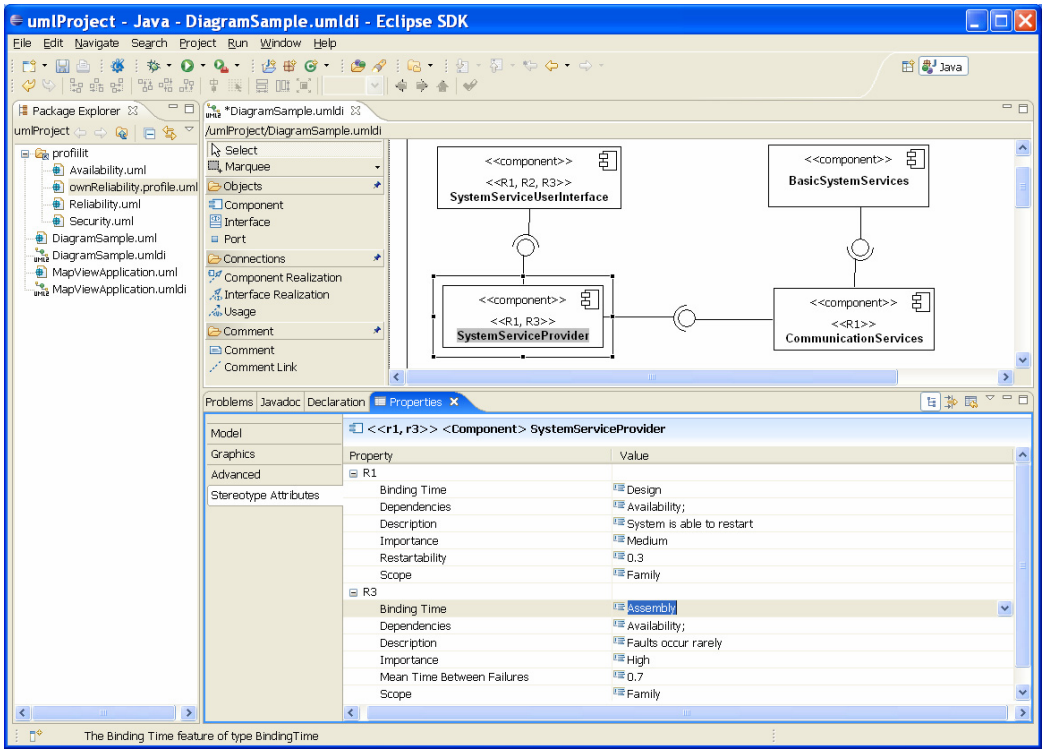


Figure 22. Architecture with quality capabilities in TOPCASED.

The properties view at the bottom of Figure 22 shows the stereotypes' content, i.e. the requirements' content, of the selected component. Therefore, the properties view shows the content of requirements R1 and R3. This view makes it possible to vary the requirement's values, i.e. the metric's value, scope and importance. Variability is implemented using features the UML tool offers, like

drop-down menus as shown in Figure 22. The requirement's description and dependencies fields are not modifiable with the UML tool, so there is no possibility of variation. There is also a field called *Binding time,* which has the alternatives *design*, *assembly*, *start-up* and *run-time*. The value for the *Binding time* field is selected in the architectural design phase because the correct value is not clear in the quality profile definition phase.

When the software architect has defined the software architecture and the quality requirements are connected to the defined architecture, i.e. to the defined UML model, the UML model can be stored in the file system as shown in Figure 13. The next phase is evaluating the quality of the stored architecture.

4.4 RAP tool under Eclipse

As mentioned in Chapter 2.4.4, it is necessary to port the RAP tool to Eclipse because there is no quality evaluation tool available. Java language and Eclipse's Plug-in Development Environment are only possible implementation techniques when porting the RAP tool to Eclipse. In Chapter 3.3.5 the Blackboard architectural pattern was selected as the architecture for the new RAP tool. The controller component is running in a loop in the Blackboard architecture and monitors the content of the Blackboard component. The controller schedules other component execution using the content of the Blackboard.

The functionality of the MarkovAnalysis, Analyses and Simulation components is similar to that in the first version of the RAP tool, but the GUI and UMLReader components differ when comparing the first implementation. The biggest difference is in the UMLReader component because the UML tool's open API was used earlier, but now the UML diagrams are read from the XML formatted files. The differences in the GUI component are not so widespread and mostly come from the difference between a standalone program and an Eclipse plug-in.

When porting the RAP tool to Eclipse was started, the TOPCASED UML tool was not available and the Omondo UML tool was selected instead. Because Omondo did not fulfil all the desired requirements, it should offer the possibility to change the UML tool at a later date. The Factory method design pattern offers

a solution for this kind of problem. The Factory method pattern defines how a new object is created, but gives subclasses to select which class is used when the object is created [45]. Figure 23 depicts the implementation of the UMLReader component in which the Factory method pattern is used.

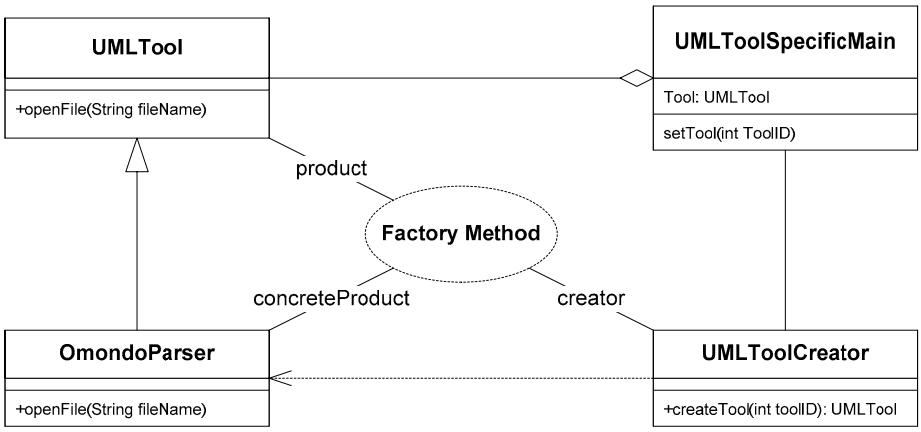


Figure 23. The Factory method applied to the UMLReader component.

As Figure 23 shows, the *OmondoParser* class extends the *UMLTool* class. Thus the *UMLToolSpecificMain* and *UMLToolCreator* classes use an interface of the *UMLTool* class and the *OmondoParser* class can be changed if needed, and the modification does not affect the other classes.

Figure 24 depicts the ported RAP tool running under the Eclipse platform. As can be seen, it is using the Omondo UML tool instead of TOPCASED.

In Figure 24 the upper part shows the simulation model of the analysed system. At the bottom is the RAP tool view, which contains the results of the analysis presented in Chapter 2.4.4. That view is divided into three parts: *Message Contents*, *Components* and *Results*. The *Message Contents* part contains a drop-down menu that contains all input messages. When the user selects a message, the message content is shown and the execution path that will be executed in that message is shown in the *Components* area. The *Results* area contains a table that shows all the components of the analysed system and their access times and fault probabilities. Finally, the fault probability for the whole system is shown in a bottom corner of the view.

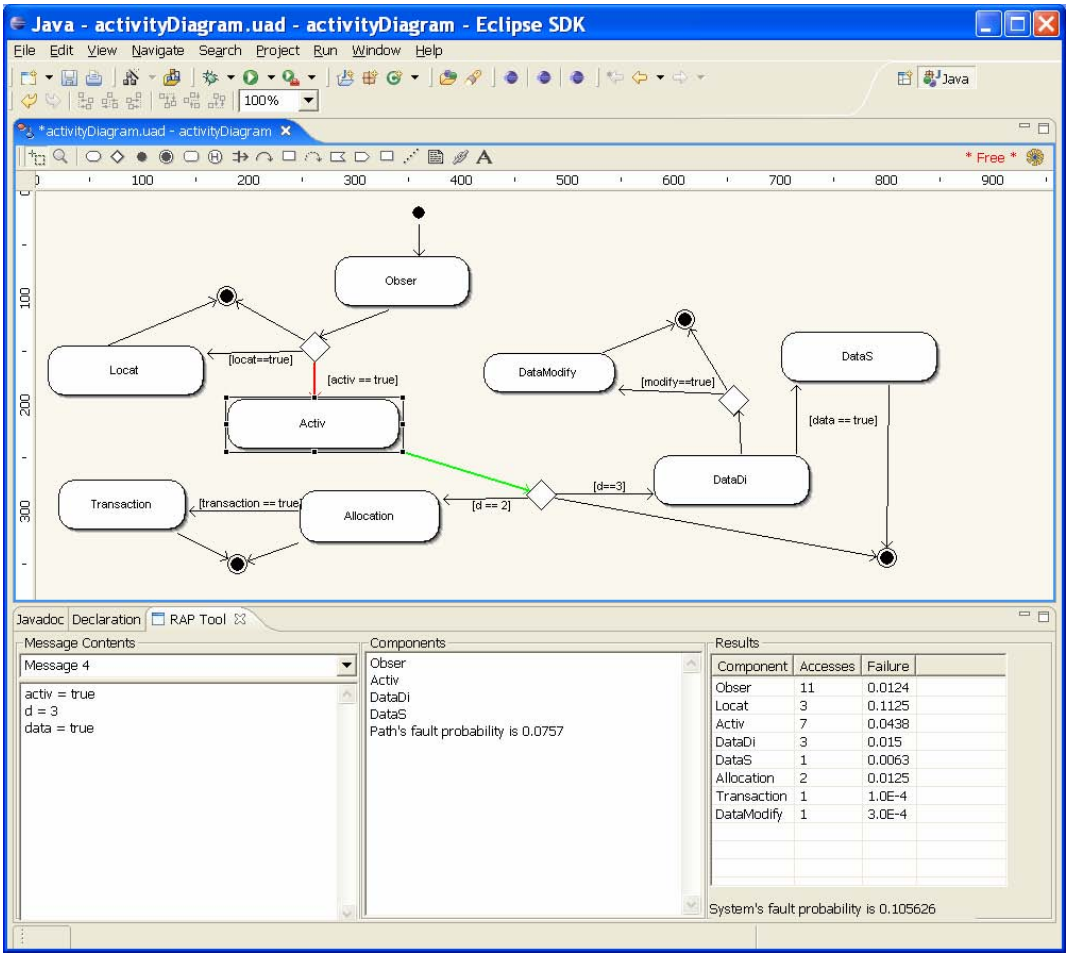


Figure 24. RAP tool under Eclipse.

As shown in Figure 14, the architecture analyser uses evaluation tools, i.e. the RAP tool in this case. The architecture analyser opens the previously defined software architecture with the RAP tool and builds a simulation model of the analysed architecture. The RAP tool utilises the defined simulation model and opened architecture to calculate the system's fault probabilities. The architecture analyser and software architect compare these calculated results with the desired software quality, and the software architect can enhance the architecture if the desired quality is not met.

4.5 Testing

The purpose of the testing process is to verify that the implemented environment contains the required functionality and meets the desired quality requirements. Therefore, the requirements that are set in Chapter 3.2, especially in Table 6 and Table 7, construct a framework for the testing process. Testing is performed in two parts: component testing and integration testing. Component testing tests each sub-system of the QO-AE separately, i.e. Protégé, Profile editor, TOPCASED UML tool and RAP tool, and integration testing tests that these parts are able to co-operate and exchange information.

The components of the Profile editor and the RAP tool are tested using the black box method; these components are depicted in Figure 17 and Figure 18. When each component is working properly, the components are connected and the programs are tested as a whole. The purpose of this procedure is to reduce programming errors and check that the programs handle the data properly.

The first requirement of the QO-AE is the requirement F1, i.e. the Eclipse plug-in. The implementation phase proved that this requirement is not met. The reason for this is that there was no suitable ontology tool available for Eclipse. However, the ontology handling process was implemented in such a way that it is possible to change the used ontology tool because it is enough that the ontology is stored in the OWL format.

The requirement F2 of the QO-AE can be directed to the ontology tool, i.e. Protégé. Protégé stores ontologies in the OWL format. The consistency of these OWL files is tested by saving the ontology in Protégé and then opening the saved OWL file in another ontology editor. The same operation is also performed vice versa. Opening the ontology outside of Protégé succeeded as well as the same process in the reverse order. Thus requirement F2 is met. This ensures that Protégé stores ontologies in a standard way, which is needed by the quality ontologies. Naturally, this testing method is not watertight but it is enough in this case.

Requirement F3 from Table 6 can be directed to the Profile editor. This requirement was tested using the reliability ontology. When the ontology was opened the Profile editor found all metrics that were stored in the ontology.

Unfortunately, other quality ontologies are not available, so the test case is quite restricted. The reliability ontology was changed a few times during the testing in order to ensure that the OWL handler was working as universally as possible. Each time the ontology was changed and then opened, the Profile editor opened it as desired. These test cases proved that requirement F3 is met.

Requirement F4 relates to the RAP tool. F4 was tested by constructing several models using the Omondo UML tool and after that opening these diagrams with the RAP tool. The RAP tool was able to open each model as desired, hence requirement F4 is reached.

The TOPCASED UML tool has to realize requirement F5. Eclipse's UML project and the TOPCASED UML tool can read the same UML diagrams, thus a diagram saved in TOPCASED should also open in Eclipse's UML project. This method was used to ensure that the UML diagrams are uniform after adding the quality profile. Different quality profiles were made using the Profile editor and these profiles were connected to the architecture models in TOPCASED. Because Eclipse's UML project was able to open these architectures with the quality requirements stored in TOPCASE, it is assumed that requirement F5 is achieved. This test also gave certainty that the UML profiles the Profile editor produced were made properly. The defined profiles contained the same information as was entered in the Profile editor and TOPCASED was able to read that information. Thus requirement F6 is also met.

The Profile editor has to implement requirements F7 and F8. These requirements were tested by entering quality requirements in the Profile editor. Some of the entered requirements contained information that was against the selected ontology. The tests proved that the Profile editor only allows requirements that correspond to the ontology. If the requirement's value is out of the metric's range, the Profile editor asks the user to give a new value or select some other metric. Consequently, requirements F7 and F8 are also reached.

The performed tests proved that all the functional requirements presented in Table 6 are achieved except for F1. The quality requirements for the QO-AE are represented in Table 7. The used design and implementation techniques ensure that the requirements Q2 integrability, Q3 interoperability and Q4 modifiability are met. Analysing whether the Q1 extensibility requirement really met is tricky

because it is impossible to define coming extension needs. However, the used interfaces were defined exactly and the existing standards were used widely. These two things should make extensions easier in the future.

Added to the requirements shown in Table 6 and Table 7 there are also the tool-specific requirements depicted in Chapters 3.2.1 and 3.2.2. When the ontology tool and the UML tool were selected, the requirements for these tools were tested. Protégé fulfils all the desired requirements but it is not an Eclipse plug-in, as mentioned earlier. The selected UML tool, i.e. TOPCASED, fulfils all the requirements, as mentioned in Chapter 3.2.2.

Chapter 3.2.3 contains the tool-specific requirements for the ported RAP tool. As mentioned earlier, the RAP tool is able to open UML diagrams made with the Omondo UML tool, i.e. requirement F4. The requirement that the calculated results are correct was tested using the same input data as in the RAP tool's first implementation. The RAP tool's new version gave the same result as the previously implemented version, so it is possible to assume that the program's logic is working correctly. In addition, the test proved that the RAP tool calculates the fault probabilities for components, execution paths and the whole system as desired. The requirement that the RAP tool is easy to port to different UML tools was not tested in practice, but the utilised design pattern promises that this will be easy.

5. Discussion

The software architecture that is described using UML models ordinarily contains the software's functional representation with no information on the quality requirements. Dropping the quality requirements out of the software architecture design process may mean that a large amount of resources have been put into building a system that does not fulfil its quality requirements [1]. The Quality-Oriented Architecting Environment is an environment implemented during this work that offers a way to manage and connect the quality requirements to the software architecture and thus improve the quality of the developed software.

5.1 Link-up to the related research

The NFR framework and the i* framework, which were described in Chapter 2 concentrates on detecting where the quality requirements originate from and refining them. These methods do not define how to connect the quality requirements to the architectural models or how to provide variability in the quality requirements between software components or software family members. The quality profile that was developed in this work makes it possible to connect the quality requirements to the architectural models. In addition, the quality profile offers a way to vary the defined requirements. The way in which these quality requirements are initially collected does not affect the quality profile.

In the RAP method, presented in Chapter 2.4.4, the reliability and availability requirements are connected to the UML model using UML profiles and tagged values. This method makes it possible to connect the requirement's dimension and its value to the model. In this work the dimension was called the metric. The RAP method contains separated profiles for the desired quality and the provided quality. The quality profile developed in this work is not restricted to depicting just the reliability and availability requirements. Instead, it makes it possible to depict any quality requirements in a UML diagram. However, the remarkable thing is that the current version of the quality profile only contains the desired quality level, not the provided quality. Showing the provided quality in the architecture is a practical matter, but this can be done using the quality profile

developed in this work – it is only necessary to add a new field for the provided quality to the quality profile.

The quality profile and the developed environment are discussed in a conference paper that has been submitted to the Software Product Lines Conference 2007 [44].

5.2 Implementation of the environment

Since the design of the QO-AE was started it has been clear that the environment should work under the Eclipse Platform. The second firm thing was that the environment should use open source software components whenever possible, rather than using commercial software components. This approach forced to broadly work with other open source software, not only Eclipse.

During the work it was realised that it was not possible to implement the whole system under the Eclipse platform as no appropriate ontology tool was available. This forced the use of a standalone ontology tool. Protégé was selected because it seemed to be the most commonly used ontology tool. However, the use of a standalone tool will be easy to fix when a suitable ontology tool becomes available for Eclipse because the implemented environment can transfer the quality ontologies using the existing file system and OWL files. Of course, problems will occur if a new ontology tool stores the ontologies using some format other than OWL, but that looks improbable at the moment. Protégé was a good selection as it offered an illustrative way to chart and develop the ontologies. In addition, Protégé offered a good tutorial on ontology development in general and by using Protégé.

The quality ontologies were read into the Profile editor using the Jena library. Jena's web pages offered many tutorials and code samples that showed how to use Jena in Java programs. The use of Jena was difficult despite the available instructions. Thus reading the ontology into the Profile editor took a great amount of time, even though the number of code lines was insignificant. The schedule for the Profile editor implementation was caught up with because constructing a UML profile was easier than was first assumed. Eclipse's UML project offered clear code samples of how to make UML profiles using Java code, and these samples were directly applicable to this case.

TOPCASED was selected as the UML tool in the QO-AE, although it is a newcomer and still needs product development. For example, not all UML 2.0 diagrams are included and the instructions are still lacking. However, TOPCASED fulfilled all the requirements within the scope of this work. The test runs proved that TOPCASED is able to open quality profiles made by the Profile editor and it can also store UML diagrams in a uniform way, as was required. In the future it will be possible to change to another UML tool, assuming the new UML tool also uses Eclipse's UML tool project.

Porting the RAP tool to the Eclipse platform was carried out as expected. The new tool produced the correct results and read the required UML diagrams. RAP tool's Eclipse version was implemented before the TOPCASED UML tool arrived, so the new RAP tool uses the Omondo UML tool. However, implementation was done in such a way that changing the UML tool will be easy.

The biggest problem during the design and implementation of the QO-AE was the lack of documentation on the open source software components. Comparing the available open source components was challenging because the only way to get accurate information on the evaluated component was to install it and use it. Anyway, using open source components offered many advantages when compared with commercial software components. For example, many open source projects having active users answered questions via mailing lists or news groups. The use of open source components also made it possible to implement the QO-AE without a need to spend money on licences for the commercial software components. These savings might be interesting in future software development.

5.3 Results

The developed environment contains open source components, i.e. Protégé and TOPCASED, and self-made Eclipse plug-ins, i.e. Profile editor and RAP tool. The quality profile the Profile editor produces was also developed during this work. The developed environment and quality profiles make it possible to take quality requirements into account as early as the architecture development. Therefore, the implemented software meets its quality requirements better, which reduces re-design work and saves money. The variation in the

requirements offered by the quality profile makes it possible to modify the quality requirements between software components and software family members.

Because the developed environment was implemented using open source components, the software industry can freely utilise the environment in the future and enhance their products. The experience with open source components that was gained during this work can be utilised in many research and industry projects in the future. Future research and development work on the developed environment is discussed in the next section.

5.4 Future research and development

The environment that was implemented during this work is the first prototype to manage quality attributes during the design of the software architecture. Thus many things need further research and development work in the future. Small-scale development relates to usability things, like the user interface of the Profile editor and the possibility to modify the existing quality profiles using the Profile editor.

At this time, the reliability ontology is the only available quality ontology and it only contains a few metrics. This restricts the use of the QO-AE. For example, the ISO's quality model that was presented in Table 1 contains many useful metrics that can be used a starting point for defining new quality ontologies. At least, security and performance ontologies are desired, in order to use of the QO-AE would be reasonable.

In the future, a reassessment of the form of the selected ontology would be appropriate, i.e. should every quality attribute have its own ontology? Alternatively, is it better to combine all the quality attributes in the same ontology? This would facilitate dependency management but the number of metrics in the Profile editor would explode. In any case, when new and wider ontologies are available, the environment's performance may cause problems because the tested reliability ontology only contains a few classes and opening it with Jena takes a few seconds. This is not critical for the utilisation of the environment but it might reduce the usability.

The first version of the designed quality profile has a dependencies field that contains the dependencies between different quality profiles, like between security and availability. This way of representing dependencies is not detailed enough because not all the security requirements affect all the availability requirements. Thus, use of the dependencies field needs enhancing in the future; showing the dependencies between individual requirements would be an appropriate way. As mentioned earlier, the quality profile does not contain information on the provided quality, and a simple solution to this problem would be to add a new field to the quality profile. When this field is added, the evaluation tools can store the results of the evaluation in it.

The weak point of the developed environment is that the new RAP tool uses a different UML tool to the other parts of the environment. This restricts efficient use of the environment. Therefore, it is necessary to port the RAP tool to use the TOPCASED UML tool as well. Using the RAP tool needs wide understanding of the evaluated system because the analyzer has to construct a simulation model of the system; automating this task in the future is necessary in order to hasten the evaluation process. In the future the RAP tool will have to write the results of the evaluation in the quality profile to enable a comparison between the required and the provided quality. Added to this, the RAP tool only offers the possibility to evaluate the architecture's reliability, so new quality evaluation tools are also needed for the future. The IEE method that was mentioned in Chapter 2.1.2 could offer one possible evaluation method. In the ideal case there would be many quality ontologies and quality evaluation tools. This would make it possible to connect all the desired quality requirements to the architecture and finally analyse whether the quality requirements were met or not.

The implemented environment was tested using a test scenario. To ensure that the environment is working perfectly needs wider test cases. Searching for a suitable test case from industry has been started because the purpose is experiment with the developed environment in a real software development situation. The QO-AE was implemented in the SVAMP project, which is a collaboration project between VTT and HUS. At the same time as the QO-AE was being implemented, HUS implemented a similar kind of environment for functional requirements. The next step is to integrate these two environments. Finally, the integrated environment will be tested using an industrial test case.

6. Conclusion

The main research problem in this work was how to represent and connect the quality requirements to the software architecture in such a way that the requirements can vary between software components and software family members. The purpose of this work was to design and implement an environment on top of the Eclipse platform.

The Quality-Oriented Architecting Environment (QO-AE) was designed and implemented during this work. This environment makes it possible to define the quality requirements using a specific quality ontology. The major contribution of this work was quality profiles, which make it possible to collect the quality requirements in a uniform way and then connect the requirements to the software architecture, i.e. UML models.

QO-AE was combined using freely available open source components and homemade software components. The environment contains four main parts: an ontology tool, a homemade Profile editor, a UML tool and a quality evaluation tool. In this work the UML, ontology and quality evaluation tools for Eclipse were evaluated. The result of the evaluation was that TOPCASED is the best open source UML tool for Eclipse as suitable ontology and quality evaluation tools are not currently available. Thus it was necessary to port the existing RAP tool to Eclipse and use a standalone ontology tool, i.e. Protégé.

The tests proved that the implemented environment works as expected. It is possible to define quality attributes to the ontology format using Protégé. These quality ontologies are utilized for defining the quality profiles, which are used to identify and define the quality requirements and represent them in the architectural models. However, evaluating the quality of designed architecture is possible after making the ported RAP tool and used architecture design tool compatible.

The main goal of this work was achieved. The contributed quality profile makes it possible to define the software's quality requirements and connect these requirements to the architectural models in a uniform way that offers varying requirements between software components. The implemented Profile editor facilitates defining these quality profiles and in the future the RAP tool's Eclipse

version will make it possible to evaluate the provided quality from the reliability viewpoint. A conference paper that discusses the results of this work has been submitted to the Software Product Lines Conference 2007.

QO-AE offers a good base for enhancing the quality of software architectures. However, the designed environment needs additional development before the software industry can utilise it.

References

[1] Bosch J. (2000) Design & Use of Software Architectures – Adopting and evolving a product-line approach. Addison Wesley, Harlow, 354 p.

[2] Bass L., Clements P. & Kazman R. (1998) Software Architecture in Practice, second edition. Addison-Wesley, Massachusetts, 528 p.

[3] Evans J. R. & Lindsay W. M. (1999) The Management and Control of Quality. South-Western College Publishing, Cincinnati, 785 p.

[4] OMG (2005) Unified Modeling Language: Superstructure version 2.0 formal/05-07-04, 694 p.

[5] Rubel D. (2006) The Heart of Eclipse. ACM Queue, Vol. 4, No. 8, pp. 36–44.

[6] Praxiom Research Group (25.10.2006) ISO 9000 DEFINITIONS. URL: http://www.praxiom.com/iso-definition.htm.

[7] IEEE (1998) IEEE Standard for a Software Quality Metrics Methodology. IEEE Std. 1061 – 1998.

[8] Chung L., Nixon B. A., Yu E. & Mylopoulus J. (2000) Non-Functional Requirements. Kluwer Academic Publishers, Boston, 439 p.

[9] Grünbacher P., Egyed A. & Medvidovic N. (2001) Reconciling Software Requirements and Architectures: The CBSP Approach. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium, August 27–31, Toronto, pp. 202–211.

[10] ISO/IEC (2001) ISO/IEC 9126-1 International Standard: Software engineering – Product quality. Part 1: Quality model. 25 p.

[11] ISO/IEC (2003) ISO/IEC 9126-2 Technical Report: Software engineering – Product quality. Part 2: External metrics. 86 p.

[12] ISO/IEC (2003) ISO/IEC 9126-3 Technical Report: Software engineering – Product quality. Part 3: Internal metrics. 62 p.

[13] Matinlassi M. & Niemelä E. (2003) The Impact of Maintainability on Component-based Software Systems. In: Euromicro conference, September 1–6, Antalya, Turkey, Vol. 29, pp. 25–32.

[14] Matinlassi M., Niemelä E. & Dobrica L. (2002) Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. VTT Publications 456, VTT Technical Research Centre of Finland, Espoo, 129 p. + app. 10 p. URL: http://virtual.vtt.fi/inf/pdf/publications/2002/P456.pdf.

[15] QADA brochure (19.1.2007) Quality-driven Development of Software Family Architectures. URL: http://virtual.vtt.fi/qada/images/qada_esite_final.pdf. 7 p.

[16] Immonen A. (2006) A method for predicting reliability and availability at the architectural level. In: Käkölä T. & Duenas J. C. (eds.) Software product lines: Research issues in engineering and management. Springer, New York. Pp. 373–422.

[17] Gruber T. (1992) A Translation Approach to Portable Ontology Specifications. Knowledge acquisition 5(2): 1993, pp. 199–220.

[18] W3C (27.10.2006) OWL Web Ontology Language: Use Cases and Requirements. URL: http://www.w3.org/TR/webont-req/.

[19] W3C (27.10.2006) RDF Primer – W3C Recommendation 10 February 2004. URL: http://www.w3.org/TR/rdf-primer/.

[20] W3C (27.10.2006) RDF Validation Service. URL: http://www.w3.org/RDF/Validator/.

[21] W3schools (7.11.2006) RDF Tutorial. URL: http://www.w3schools.com/rdf/default.asp.

[22] W3C (27.10.2006) OWL Web Ontology Language: Guide – W3C Recommendation 10 February 2004. URL: http://www.w3.org/TR/owl-guide/.

[23] Horridge M., Knublauch H., Rector A., Stevens R. & Wroe C. (2004) A Practical Guide to Building OWL Ontologies Using the Protégé-OWL Plugin and CO-ODE Tools, Edition 1.0. University of Manchester. URL: http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf.

[24] Davies J., Studer R. & Warren P. (2006) Semantic Web Technologies: trends and research in ontology-based systems. John Wiley & Sons Ltd., Chichester, 312 p.

[25] Denny M. (read 6.11.2006) Ontology building: A survey of editing tools. URL: http://www.xml.com/pub/a/2002/11/06/ontologies.html.

[26] Denny M. (read 6.11.2006) Ontology Tools Survey, Revisited. URL: http://www.xml.com/pub/a/2004/07/14/onto.html.

[27] Protégé (6.11.2006) URL: http://protege.stanford.edu/.

[28] SWOOP (6.11.2006) URL: http://www.mindswap.org/2004/SWOOP/.

[29] TopBraid (6.11.2006) URL: http://www.topbraidcomposer.com/.

[30] Jena (6.11.2006) URL: http://jena.sourceforge.net/.

[31] OMG (1.11.2006) Object Management Group, URL: http://www.omg.org/.

[32] Koskimies K., Koskinen J., Maunumaa M., Peltonen J., Selonen P., Siikarla M. & Systä T. (2004) UML Työvälineenä ja tutkimuskohteena. Tietojenkäsittelytiede, No. 21, pp. 19–51.

[33] The Eclipse foundation (24.10.2006) Eclipse.org. URL: http://www.eclipse.org/.

[34] Gamma E. & Beck K. (2004) Contributing to Eclipse: Principles, Patterns and Plug-Ins. Addison-Wesley, Boston, 395 p.

[35] Eclipse Platform Technical Overview (24.10.2006) URL: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[36] The Eclipse Foundation (25.10.2006) Eclipse 3.2 Documentation URL: http://help.eclipse.org/help32/index.jsp.

[37] Gruber O., Hargrave B. J., McAffer J., Rapicault P. & Watson T. (2005) The Eclipse 3.0 Platform: Adopting OSGi technology. IBM Systems journal, Vol. 44, No. 2, pp. 289–299.

[38] The Eclipse Foundation (25.10.2006) SWT: The Standard Widget Toolkit. URL: http://www.eclipse.org/swt/.

[39] Eclipse-Plugins.info (10.6.2006) URL: http://eclipse-plugins.2y.net/.

[40] Niskanen A. (2005) Työkalu luotettavuuden mallipohjaiseen analysointiin. Master thesis. University of Oulu, Electrical and Information Engineering, Oulu.

[41] Sparx Systems Enterprise Architect URL: http://www.sparxsystems.com/.

[42] Zhou J. & Niemelä E. (2007) OntoArch Ontology and Ontology-based Reliability-aware Architecture Design and Evaluation. Submitted to: Journal of systems and software.

[43] Buschmann F., Meunier R., Rohnert H., Sommerland P. & Stal M. (1996) Pattern-oriented Software Architecture: a System of Patterns. John Wiley, Chichester, 457 p.

[44] Evesti A., Niemelä E. & Zhou J. (2007) Quality Variability in Architecture Design. Submitted to the Software Product Line Conference 2007, 10 p.

[45] Gamma E., Helm R., Johnson R. & Vlissides J. (1994) Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison Wesley, 395 p.

Series title, number and report code of publication

VTT Publications 636
VTT-PUBS-636

Author(s)
Evesti, Antti

Title

Quality-oriented software architecture development

Abstract

Producing software products of good quality requires that quality requirements are taken into account as early as possible. In theory, the first place in which quality requirements can be addressed is architectural models of software. However, in practice, the software's architecture is only used to describe the functionality of the developed software. This means that the implemented software may not fulfil its quality requirements and some parts of the implementation process might be useless. The main research problem in this work is how to define and connect quality requirements with the software architecture in such a way that the requirements can vary between software components and software family members.

An environment for defining and collecting software's quality requirements was designed and implemented in this work. The environment consists of three main parts: quality meta-data management, quality modelling and quality evaluation. The quality meta-data management provides a possibility to define each quality attribute into an ontology form. These ontologies are utilized when quality requirements are defined in order to define the requirements in a uniform way in the quality modelling phase. Quality requirements are defined according to the UML profile developed for that purpose, so that it would be possible to represent these requirements in the architectural models. Finally, the architecture's quality is evaluated using evaluation tools. The purpose was to implement the whole environment on the Eclipse platform using available open source components. The Eclipse was selected because it is a widely used open source platform, which makes it easier to distribute the software developed during this work. The Eclipse tool evaluation confirmed that TOPCASED is the best available UML tool for the Eclipse and the best Eclipse ontology tool is EODM. However, EODM does not fulfil all the desired requirements of this work and this enforced the use of the Protégé ontology tool.

The implemented environment was tested using the defined test scenarios. The tests proved that the implemented environment works as expected. In addition, the developed quality profile offered an appropriate way to connect the defined quality requirements to the architectural models. There are still many things that need additional research and development before the environment and the quality profile can be utilized in software design in industry.

ISBN
978-951-38-7011-9 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN	Project number
VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	562

Date	Language	Pages
April 2007	English, Finnish abstr.	79 p.

Name of project	Commissioned by
SVAMP	Tekes – the Finnish Funding Agency for Technology and Innovation, VTT Technical Research Centre of Finland

Keywords	Publisher
quality-oriented software architecture, software development, quality requirements, ontologies, quality meta-data management, quality modelling, quality evaluation, Unified Modeling Language	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 Publications 636
VTT-PUBS-636

Tekijä(t)
Evesti, Antti

Nimeke

Laatuohjattu ohjelmistoarkkitehtuurisuunnittelu

Tiivistelmä

Laadukkaiden ohjelmistotuotteiden valmistaminen edellyttää, että laatuvaatimukset on huomioitu mahdollisimman aikaisessa vaiheessa. Teoriassa ensimmäinen vaihe, jolloin laatuvaatimukset voidaan osoittaa, on ohjelmistoarkkitehtuuri. Käytännössä ohjelmistoarkkitehtuurilla kuvataan suunniteltavan ohjelmiston toiminnallisuutta. Tämän vuoksi valmis ohjelmisto ei välttämättä täytä asetettuja laatuvaatimuksia, joten osa toteutuksesta voi olla käyttökelvotonta. Pääongelmana tässä työssä on se, kuinka määritellä ja liittää laatuvaatimuksia ohjelmistoarkkitehtuuriin siten, että vaatimukset voivat vaihdella ohjelmistokomponenttien ja ohjelmistotuoteperheiden välillä.

Tässä työssä suunniteltiin ja toteutettiin ympäristö ohjelmiston laatuvaatimuksien määrittelyyn ja keräämiseen. Toteutettu ympäristö koostuu kolmesta osasta: laadun määrittelystä, laadun mallintamisesta ja laadun arvioinnista. Laadun määrittelyssä jokainen laatuattribuutti määritellään ontologiamuotoon. Ontologioiden avulla laatuvaatimukset määritellään yhtenäisellä tavalla laadun mallintamisvaiheessa. Laatuvaatimukset määritellään tätä tarkoitusta varten kehitetyn UML-profiilin mukaisesti, jotta vaatimukset voidaan esittää arkkitehtuurimalleissa. Lopuksi arkkitehtuurin laatua arvioidaan arviointityökaluilla. Tarkoituksena oli toteuttaa koko ympäristö Eclipse-alustalle hyödyntäen saatavilla olevia avoimen lähdekoodin komponentteja. Eclipse valittiin, koska se on laajasti käytetty avoimen lähdekoodin alusta, mikä mahdollistaa työssä kehitettävän ohjelmiston helpon levittämisen. Toteutettu Eclipse-työkaluarviointi osoitti, että TOPCASED on paras saatavilla oleva UML-työkalu Eclipselle ja paras Eclipsen ontologiatyökalu on EODM. EODM ei kuitenkaan täyttänyt kaikkia tämän työn vaatimuksia, joten jouduttiin käyttämään Protégé-ontologiatyökalua.

Toteutettu ympäristö testattiin käyttäen määriteltyjä testitapauksia. Testit osoittivat, että toteutettu ympäristö toimii oletetulla tavalla. Lisäksi kehitetty laatuprofiili tarjosi tarkoituksenmukaisen tavan yhdistää määritetyt laatuvaatimukset arkkitehtuurimalleihin. Tarvitaan kuitenkin lisää tutkimusta ja tuotekehittelyä ennen kuin ympäristöä ja laatuprofiilia voidaan hyödyntää ohjelmistosuunnitteluun teollisuudessa.

ISBN
978-951-38-7011-9 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN	Projektinumero
VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	562

Julkaisuaika	Kieli	Sivuja
Huhtikuu 2007	Englanti, suom. abstr.	79 s.

Projektin nimi	Toimeksiantaja(t)
SVAMP	Tekes, VTT

Avainsanat	Julkaisija
quality-oriented software architecture, software development, quality requirements, ontologies, quality meta-data management, quality modelling, quality evaluation, Unified Modeling Language	VTT PL 1000, 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374

VTT PUBLICATIONS

620 Talja, Heli. Asiantuntijaorganisaatio muutoksessa. 2006. 250 s. + liitt. 37 s.

621 Kutila, Matti. Methods for Machine Vision Based Driver Monitoring Applications. 2006. 82 p. + app. 79 p.

622 Pesonen, Pekka. Innovaatiojohtaminen ja sen vaikutuksia metsäteollisuudessa. 2006. 110 s. + liitt. 15 s.

623 Hienonen, Risto & Lahtinen, Reima. Korroosio ja ilmastolliset vaikutukset elektroniikassa. 2007. 243 s. + liitt. 172 s.

624 Leviäkangas, Pekka. Private finance of transport infrastructure projects. Value and risk analysis of a Finnish shadow toll road project. 2007. 238 p. + app. 22 p.

625 Kynkäänniemi, Tanja. Product Roadmapping in Collaboration. 2007. 112 p. + app. 7 p.

626 Hienonen, Risto & Lahtinen, Reima. Corrosion and climatic effects in electronics. 2007. 242 p. + app. 173 p.

627 Reiman, Teemu. Assessing Organizational Culture in Complex Sociotechnical Systems. Methodological Evidence from Studies in Nuclear Power Plant Maintenance Organizations. 2007. 136 p. + app. 155 p.

628 Kolari, Kari. Damage mechanics model for brittle failure of transversely isotropic solids. Finite element implementation. 2007. 195 p. + app. 7 p.

629 Communications Technologies. VTT's Research Programme 2002–2006. Final Report. Ed. by Markku Sipilä. 2007. 354 p.

630 Solehmainen, Kimmo. Fabrication of microphotonic waveguide components on silicon. 2007. 68 p. + app. 35 p.

631 Törrö, Maaretta. Global intellectual capital brokering. Facilitating the emergence of innovations through network mediation. 106 p. + app. 2 p.

632 Lanne, Marinka. Yhteistyö yritysturvallisuuden hallinnassa. Tutkimus sisäisen yhteistyön tarpeesta ja roolista suurten organisaatioiden turvallisuustoiminnassa. 2007. 118 s. + liitt. 81 s.

633 Oedewald, Pia & Reiman, Teemu. Special characteristics of safety critical organizations. Work psychological perspective. 2007. 114 p. + app. 9 p.

634 Tammi, Kari. Active control of radial rotor vibrations. Identification, feedback, feedforward, and repetitive control methods. Espoo 20076. 151 p. + app. 5 p.

636 Evesti, Antti. Quality-oriented software architecture development. 2007. 79 p.

Julkaisu on saatavana	Publikationen distribueras av	This publication is available from
VTT PL 1000 02044 VTT Puh. 020 722 4404 Faksi 020 722 4374	VTT PB 1000 02044 VTT Tel. 020 722 4404 Fax 020 722 4374	VTT P.O. Box 1000 FI-02044 VTT, Finland Phone internat. + 358 20 722 4404 Fax + 358 20 722 4374

ISBN 978-951-38-7011-9 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)