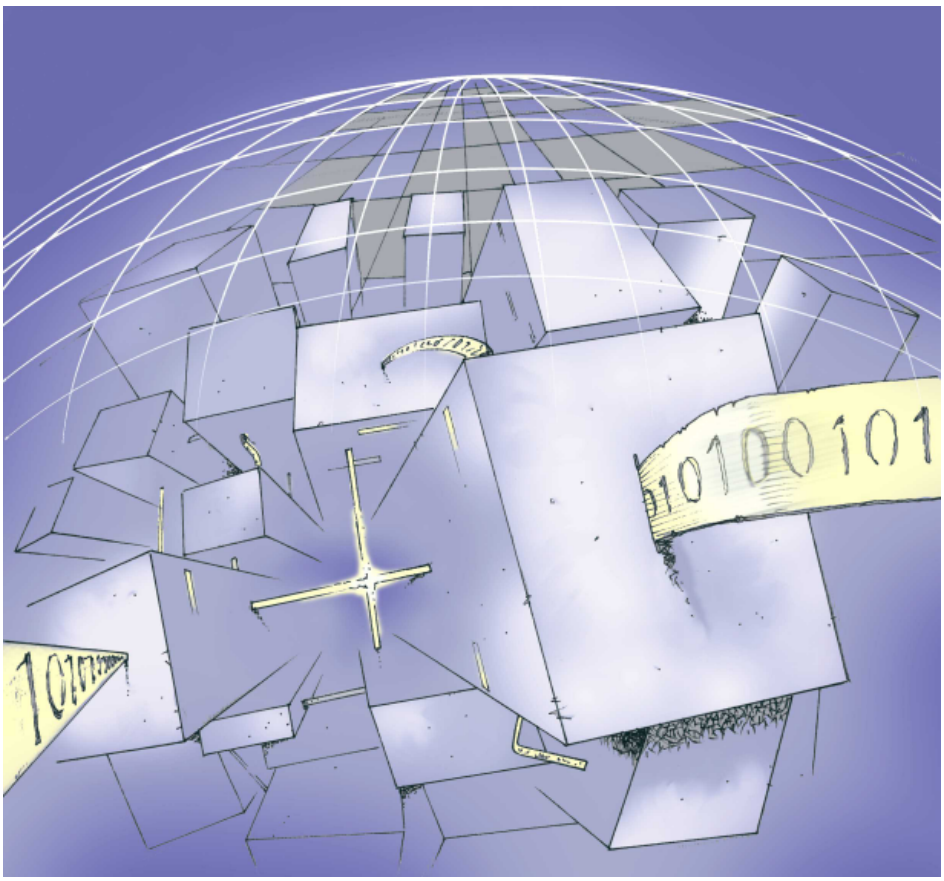
ESPOO 2002 VTT PUBLICATIONS 484



Anne Taulavuori

Component documentation in the context of software product lines

VTT PUBLICATIONS 484

Component documentation in the context of software product lines

Anne Taulavuori
VTT Electronics

ISBN 951–38–6021–3 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6022–1 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2141, faksi (08) 551 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2141, fax (08) 551 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2141, fax + 358 8 551 2320

Technical editing Marja Kettunen

Otamedia Oy, Espoo 2002

Taulavuori, Anne. Component documentation in the context of software product lines. Espoo 2002. VTT Publications 484. 111 p. + app. 3 p.

Keywords component documentation, software product lines, software engineering, component documentation pattern

Abstract

The use of third-party components in software system development is rapidly increasing. The product lines have also adopted this new tendency, as the COTS and OCM components are increasingly being used in product-line-based software engineering. Component documentation has become a key issue in component trading because it often is the only way of assessing the applicability, credibility and quality of a third-party component, especially for product lines in which the common architecture determines the decisive requirements and restrictions for components. However, at the present time there is no standard model for component documentation, and, therefore, the component documents are often inconsistent, insufficient and of various quality. The lack of a standard documentation model is thus one of the bottlenecks in component trading.

The purpose of this thesis is to define the documentation requirements of software components and form a standard documentation pattern from these requirements. The documentation requirements are examined from the viewpoint of the software product lines, where the common product line architecture may define several specific requirements for a component. The standard pattern is a skeleton of documentation, defining the content and structure for component documentation. The pattern ensures the documentation that assists the integrator in successful component selection, validation, integration and use within product lines. The development of the documentation is defined by identifying the roles responsible for the documentation and associating them with the pattern.

Definition of the documentation pattern is not sufficient for the adoption of a new documentation practice. An environment that supports the development of documentation is also required. This thesis also introduces the developed documentation system, which defines how the component documentation could be implemented. The system provides guidelines concerning how to document a

software component. It also offers the tools and technology for the development and handling of documents, and ensures that the developed documentation is in accordance with the pattern. In addition, the system is also applicable when the development of the documentation is split between different organisations. An evaluation of the documentation pattern is presented at the end of this thesis.

Preface

Software components and documentation were examined in several projects at VTT Electronics. The development of the component documentation pattern was started in the EMU project during the summer of 2000. The roles of the persons responsible for the documentation were defined and directed to the pattern. The possibilities and advantages of the use of the XML technology in component documentation were also examined and preliminarily tested.

The development of the documentation tools was started in the Codo project during the autumn of 2000. The Codo project was carried out at VTT Electronics as a programming project of the Information Processing Science Department of the university of Oulu. The result of the project was an extension to a commercial CASE tool.

The implementation of the documentation system was studied and tested in the PLA programme during in the first half of 2001. The component documentation system was developed and implemented, and the documentation pattern refined, as a result of the project. The documentation system was validated by documenting an example component using the system.

The refinement of the documentation pattern was continued in the Minttu project in spring of 2002. The component documentation pattern was refined to notice the documentation requirements of product lines for components. The pattern was also evaluated in a practice analysis.

The writing of this thesis took place during the Minttu project and was completed in autumn of 2002. I would like to thank Research Professor Eila Niemelä from VTT Electronics and Professor Veikko Seppänen from the University of Oulu for their guidance during this thesis. I would also like to thank my fiancé Marko Immonen for his support during the writing of this thesis.

Oulu 9.10.2002

Anne Taulavuori

Contents

List of abbreviations

API	Application Programming Interface
CASE	Computer-Aided Software Engineering
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
DCOM	Distributed Component Object Model
DDS	Data Distribution Service
DOM	Document Object Model
DTD	Document Type Description
ESA	European Space Agency
FAQ	Frequently Asked Questions
FOSI	Formatting Output Specification Instance
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ID	Identification
IE	Internet Explorer
IEEE	Institute of Electrical and Electronics Engineers

IPv6	Internet Protocol version 6
ISO/IEC	International Standard Organization / International Electrotechnical Commission
JDK	Java Development Kit
JDOM	Java Document Object Model
LOC	Lines Of Code
MOTS	Modified Off-The-Shelf
MSR	MicroSoft Research
MSXML	MicroSoft XML
N/A	Not Applicable
NPLACE	National Product Line Asset CEnter
OCM	Original software Component Manufacturing
OS	Open Source
OTSO	Off-The-Shelf Option
PL	Product Line
PLA	Product Line Architecture
PORE	Procurement-Oriented Requirements Engineering
RMI	Remote Method Invocation
SAP	Service Access Protocol

SAX	Simple API for XML
SEI	Software Engineering Institute
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
UML	Unified Modelling Language
W3C	World Wide Web Consortium
WWW	World Wide Web
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSL	eXtensible Style Language
XSLT	XSL Transformation

1. Introduction

Third-party software components, generic or partially tailored, are increasingly used in software product lines. Modern software systems are thus based on integrated components (Niemelä et al. 2002). A software product line is a collection or family of software systems with similar or overlapping functionality and a common architecture that satisfies the mission requirements for that set of systems (NPLACE 2002). The product line architecture is explicitly designed and assessed, offering a common core for each product of the product family (Bosch 2000). Components are used as building blocks in product development.

Third-party components are expected to provide a product integrator with a lot of benefits, such as costs savings, improved productivity and faster time-to-market (Meyers & Oberndorf 2001). The third-party components are here defined to include Commercial Off-The-Shelf (COTS) components, Original software Component Manufacturing (OCM) components, Modified Off-The-Shelf (MOTS) components and Open Source (OS) components. COTS components have been sold on the market for several years, whereas the use of OCM, MOTS and OS components is increasing in today's and the future's software systems. COTS components are ready-made components and are bought "as such" (Brownsword et al. 2000). OCM components are developed in collaboration with the component integrator and the provider (Seppänen et al. 2001). MOTS components are tailored to buyer-specific purposes from COTS components (IEEE 1993). OS components are offered as free source components (Open Source Initiative 2002).

In product-line-based software engineering the integration of components is controlled by the product line architecture. The architecture is the main asset in product lines, and, therefore, it affects decisively on component-related decisions by determining the main capabilities for the required components. Traditionally, requirements for components are classified to functional and quality requirements (ESA 1991). However, particular requirement specification standards cannot easily be used here for defining components because in product lines the components are intended to be used on a wider scale than with software developed for a specific purpose.

In product lines the purpose of each component is defined in the context of the product line architecture. The selected third-party components not only have to satisfy the functional and quality requirements but also have to be applicable to the product line architecture itself. The architectural rules of the product line define several architecture-specific requirements for components, such as quality and communication (Bosch 2000). Quality is particularly important in product lines because an unwanted property or a side effect can cause higher additional expenses than the use of the component in a single system (Niemelä et al. 2002).

When using the third-party components, the product line integrator has to be able to assess the applicability of the components to the requirements and to the product line architecture. Component documentation is the key issue in component trading because proper documentation is often the only way for integrators to assess the applicability of a component, as well as its credibility and maturity. Documentation ensures the successful component selection, validation and integration for product lines and assists the integrator in using the component as intended.

At the present time there is no standard model for software component documentation. Components have been traditionally documented as a part of software systems, which is not appropriate for third-party components that are intended for use in several contexts. A self-contained documentation model is thus required. None of the recent component documentation models are appropriate for third-party components because they mainly concentrate on component reuse inside an organisation. Furthermore, they do not consider the possible division of component development between the component integrator and the component provider. The lack of a standard documentation model has meant that every component provider has documented his components according to his own practices. The consequences are that the documents are inconsistent, insufficient and of various quality, and the required component information may be missing or is difficult to find.

1.1 Purpose of the thesis

The importance of software component documentation was noted in spring of 2000 in a project, which examined the development needs of component

software. The component documentation was seen as a bottleneck in the component development, acquirement and utilisation processes (Niemelä et al. 2000). The specified form in describing component capabilities was defined to be a prerequisite for component trading. The component integrator has to be able to validate the component properties before making the buying decision.

This thesis focuses on examining the documentation requirements of software components from the viewpoint of the product line integrator. The integrator is a component buyer who builds products using components. The components are considered to be third-party components, but the same requirements can be applied to in-house components as well. The aim is to find out what the component integrator needs to know about the component to be able to search for the component, select it from among the candidates, validate it, integrate it with his product line, and use and maintain it within the product line. The defined information requirements are transformed into component documentation requirements. The product line architecture's impact on documentation requirements is examined closely. This thesis also examines responsibilities in documentation, and how the documentation of components could be implemented so that the documentation work can be predetermined and not require much extra work by the provider.

The result of this thesis is a standard documentation pattern for components. The pattern defines all the required information about a component that the documentation must include, and also the structure of the documentation. The roles responsible for each set of information are identified and associated with the pattern. In addition, this thesis introduces a developed component documentation system that enables the development of documentation side by side with the component development, and ensures that the documentation follows the defined pattern. The system thus ensures the consistency and quality of the component documentation. One of the aims in the development of the documentation system was to evaluate the maturity of the selected technology and tools.

This thesis also provides an analysis of the documentation pattern. The pattern is analysed in three different ways: developing documentation for a component using the documentation system, interviewing the industrial software engineers,

and comparing the pattern with the documentation of four commercial components**.**

1.2 Research problem and methods

The research problem addressed in this thesis can be stated as follows:

What are the documentation requirements of a software component and how should component documentation be implemented?

This research involves both theoretical and empirical aspects. Figure 1 describes the flow of the research.

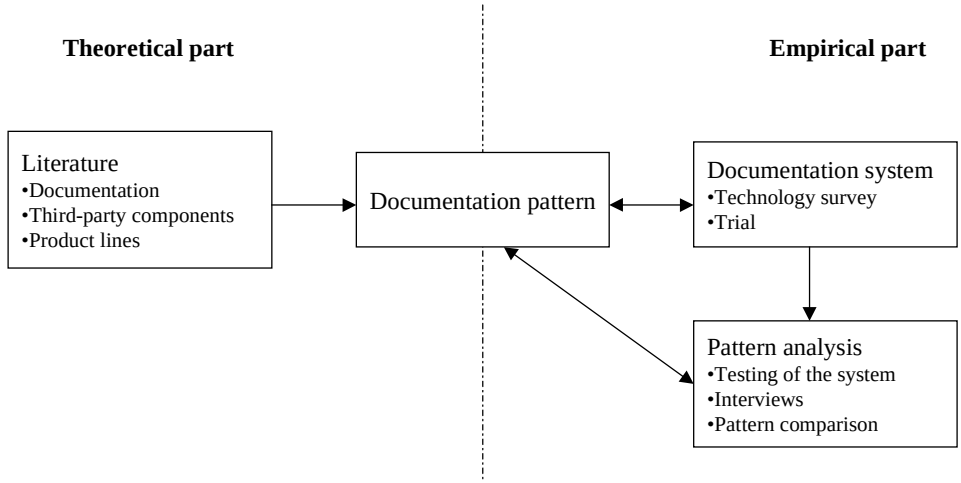


Figure 1. The research path followed.

The theoretical part of the research denotes analytical study of the literature that concentrates on documentation, third-party components and product lines. The component's concepts are examined and some previous component documentation models are studied. The purpose is to determine which properties of a component the previous documentation models emphasise. The use of components in the context of product lines is defined in order to achieve the product line integrator's viewpoint on components. The aim is to define the

critical properties of the component that must be identifiable in order to be able to assess the applicability of component for product lines.

The defined critical properties of the component are translated into component documentation requirements. The requirements are then organised in a logical sequence, and the component documentation pattern developed.

The empirical part of this thesis concentrates on pattern verification and analysis. The development of components and some documentation tools and technologies are examined in order to determine the implementation alternatives for the environment that supports the development of the documentation. Tools and technologies are selected to implement a documentation system. An example component is documented with the documentation system to evaluate the pattern from the component provider's viewpoint. Software engineers that are experienced in the use of components are interviewed to achieve a component integrator's view of the pattern. The pattern is also compared to the documentation of selected COTS and OS components to ascertain how it differs from recent documentation practices. The developed pattern is further refined in consequence of the analysis.

1.3 Limitations of the study

The limit between a component and an application is sometimes ambiguous. An application is independent software that can be used as such. According to the IEEE (1990) definition, a component is one of the parts that make up a system. Therefore, components are not independent applications but they are used to build applications. Generally, software components can be defined as including anything between a simple algorithm and a subsystem (McClure 1994). In this thesis components are seen as nearly independent pieces of software that fulfil a clear function, are used to build larger applications and can be reused in different contexts.

In component trading the documentation of software components has been seen as a key issue (Niemelä et al. 2000). This thesis concentrates on the documentation requirements of third-party software components – i.e. what the component integrator needs to know about the component in order to be able to

select it from several candidates, integrate it into his own system and use it as intended. The component management issues, acquisition processes and business contracts that are involved in component trading are not discussed here. Instead, the component selection is examined, because documentation has a great impact on this activity.

The developed documentation pattern that the thesis introduces is best applied to medium-size components. The size of a component or a piece of software has been traditionally defined by the number of physical lines of code, such as LOC (Lines Of Code) (Park 1992; Pressman 1997). In this thesis medium-size components are seen as average components that are not the most simple, like specific algorithms or libraries, but have some complexity. Niemelä (1999) classifies components into primary components for applications, building-blocks for applications and logical subsystems for system-integration. According to this, medium-sized components can be defined as building blocks for larger subsystems or applications that use primary components to achieve their actions. The sale of the smaller components may not always be economically profitable. Furthermore, there may not be enough markets for the larger components as the cost of reusing a component may exceed the cost of developing the component from scratch (Bosch 2000).

1.4 Structure of the thesis

The structure of this thesis is shown in Figure 2.

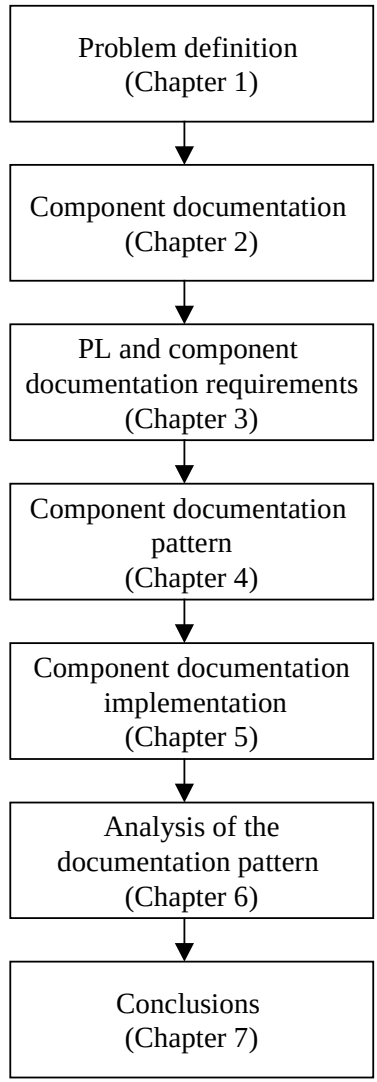


Figure 2. The structure of the thesis.

Chapter 1 provides an introduction to the subject of this thesis. In Chapter 2 the software documentation issues are discussed generally. In addition, components are discussed more precisely and some previous component documentation models are introduced.

Chapter 3 describes the use of components in the context of product lines and identifies the essential properties of components from the viewpoint of product lines. Some characteristics of a software system that integrates third-party components are also discussed. At the end of the chapter, the defined properties of components are gathered together and defined as component documentation requirements.

The developed documentation pattern for components is introduced in Chapter 4. The chapter describes all the information fields of the documentation and the document structure. The pattern is illustrated with the help of an example component, as the example component is documented following the developed pattern.

Chapter 5 introduces the implementation of a component documentation system. The use of the tools and technologies included in the system is described.

Chapter 6 describes how the documentation pattern was applied. The viewpoints of the component provider and component integrator on pattern are presented, and the comparison with selected documentation examples is performed. Chapter 7 summarises the thesis.

2. Documenting software and components

Nowadays, software components are not documented in any standard way. A common custom has been to document components as a part of the software systems in which they are used (IEE 1986; Ogush et al. 2000; Sommerville 1992). However, some attempts have recently been made to create a general component documentation model. At next, traditional software documentation is discussed first, after which the study concentrates on components, including the concept of component and some recent documentation models proposed for components.

2.1 Introduction to software documentation principles

Documentation has several purposes in software development. It provides users and developers a way for communicating and increases understanding by making the process and results of the software development visible. The most common meaning of documentation is to gather the requirements, designs and other deliverables of software development for use, follow-up and repository.

Traditionally, software documentation has been classified under two categories: process documentation and product documentation (Sommerville 1992). Process documentation concentrates on describing the software development process, including plans, reports, organizational standards and project standards. The main purpose of these documents is to make the software development process visible. The purpose of product documentation is to describe the delivered software product.

Product documentation is introduced here in more details. The quality and handling of documents is also discussed.

2.1.1 Product documentation

Product documentation can be divided into system documentation and user documentation (Sommerville 1992). System documentation follows the software design model, such as the waterfall, spiral or incremental model. All the design

models include some common design phases, such as analysis, design, implementation and testing. The order and repetition of the phases vary in different design models, but the purpose of phases is the same: to produce documents that are used as an input for the ensuing design phase.

The common system documents developed for software can be defined as including a system specification, a software requirements specification, a design specification, a source code and a test specification (Pressman 1997). Figure 3 describes the phases in which the documents are developed in the waterfall model of software development.

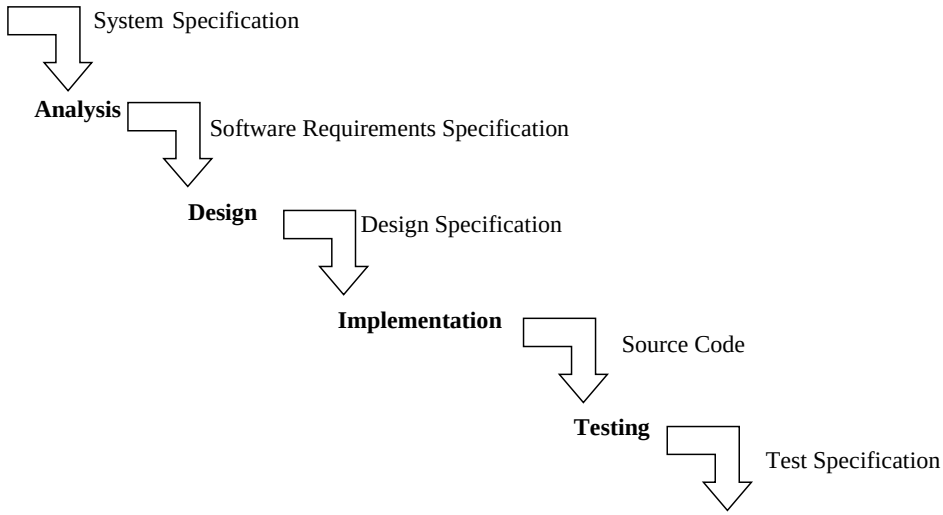


Figure 3. Common software design phases and documents.

System specification is the foundation for hardware, software, database and human engineering. It defines the requirements for all the system elements that the software is part of and then allocates some subset of the requirements to the software (Pressman 1997). Thus the system specification describes the functions, quality requirements and constraints for the software.

The software requirements specification refines the functions and quality allocated to the software and constructs a detailed functional and behavioural description, an indication of the quality requirements and design constraints, and

other data pertinent to the requirements (IEEE 1990; Wallace et al. 1992). The purpose of the software requirements specification is to describe the software itself and the relationship between the software component and the rest of the system. The specification should also provide the validation criteria that help to recognise the successful implementation of the software, providing performance bounds, classes of test and expected software response (Pressman 1997).

The design specification documents the data design, architectural design, interface representation and procedural details (Pressman 1997; Wallace et al. 1992). Data design defines the data objects and resultant data, file and database structures. Architecture design presents the structure charts that indicate the program architecture. The static structure of the architecture should be described in terms of its logical components and their interconnections. Ogush et al. (2000) suggest a template that can be used to document each component of the architecture (Table 1).

Table 1. Component specification template (Ogush et al. 2000).

Component	A unique identifier for the component.
Responsibilities	Responsibilities, provided interfaces, and rationale.
Collaborators	Other components that the component interacts with.
Notes	Information about multiplicity, concurrency, persistency, parameterisation, etc.
Issues	List of issues that remain to be resolved.

The interfaces description describes the detailed external and internal program interfaces and human-machine interfaces. Procedural details define the procedural function for each software module.

The source code is a listing of the program code. To be more readable for humans, the code should be documented, or the supporting documentation for the source code that describes the problems and solutions that the code presents should be provided (Wallace et al. 1992).

The test specification documents an overall plan for the integration of the software. The test plan describes the strategy for integration, which includes a

description of the test phases, builds, schedule, environment and resources (Pressman 1997). The test procedures are documented so that the order of integration, unit tests for modules, test environment and test data are defined for each build of software. The actual test results and possible problems during the testing must be recorded at the end of the test specification. This data is vital for the software maintenance.

The purpose of the user documentation is, according to IEEE (1990), to describe to users the way in which the software system or component is to be used in order to obtain the desired results. Users can be defined as including both end users and system administrators. The user documentation usually includes a functional description, an installation document, an introductory manual, a reference manual and a system administrator's guide (Sommerville 1992). The functional description provides an overview of the system, helping the user decide whether or not the system fits his needs. The installation manual describes how to install the system in a particular environment. The introductory manual is an informal introduction to the system, advising the users to use the common system facilities. The reference manual describes the details of all the system facilities and their usage, and gives instructions on how to handle error situations. Finally, the system administrator's guide advises the system administrator on how to operate and maintain the system.

2.1.2 Document quality

The importance of software documentation has often been ignored and the documents may be badly written, difficult to understand, out-of-date and incomplete. Still, without proper documentation, the utility of the software system is degraded. The means of achieving good document quality are, for example, document design, standards and quality assurance processes (Sommerville 1992).

Document structures should be designed in advance. The document content often determines the structure. General structuring principles include the project, document and author identification, type of document, configuration management, quality assurance information and intended recipients of the document. The contents of the documents should be organised in chapters,

sections and subsections, which should be numbered. A technical document should also include an index and glossary to increase the intelligibility of the document.

Document standards can be divided into process, product and interchange (Sommerville 1992). Process standards define the process that produces documents of high quality. Product standards apply to documents produced in the software development process. These can be defined as including document identification standards, document structure standards, document presentation standards and document update standards. Interchange standards define the conventions for using standard documentation tools and thus allow documents to be transferred electronically.

Review is a quality assurance mechanism that signifies examining the documentation associated with the software system to find problems and inconsistencies. All software documents should be reviewed before approval. Design or program inspections detect detailed errors and ensure that the standards have been followed. Management reviews are intended to provide information for management about the overall progress of the software project and ensure that the plan has been followed. A technical analysis of the product documentation to find mismatches between specification, design, code or documentation is carried out in the quality review.

2.1.3 Handling of documents

For every document developed during a software development project, the document version and status must be identifiable. Version control helps to manage different versions of documents. The version number of a document consists of two parts: Version.Revision. The first part is always zero until the document has been approved for the first time. After approval the document attains the version number 1.0. After that the version number increases every time the document is edited. The first part of the version number is only raised when making a remarkable change in the document. The document has to include the change history, which includes all the drafts and approvals.

The status of the document reflects the quality of the document. The following statuses are often used: draft, proposal or approved. Only the approved documents are usually saved as official documents of a project or software. The document is released after approval.

Documents can also have variants, which are produced when small differences between software products are created. Document status, versions and variants are described in Figure 4. Variants can be, for example, the product with colour display and the product with monochrome display (Pressman 1997).

Document management may also include practices for the approval and distribution of administrative project documents and instructions for post-project filing. A document becomes a baseline after approval. A baseline is a configuration management concept that serves as basis for further development.

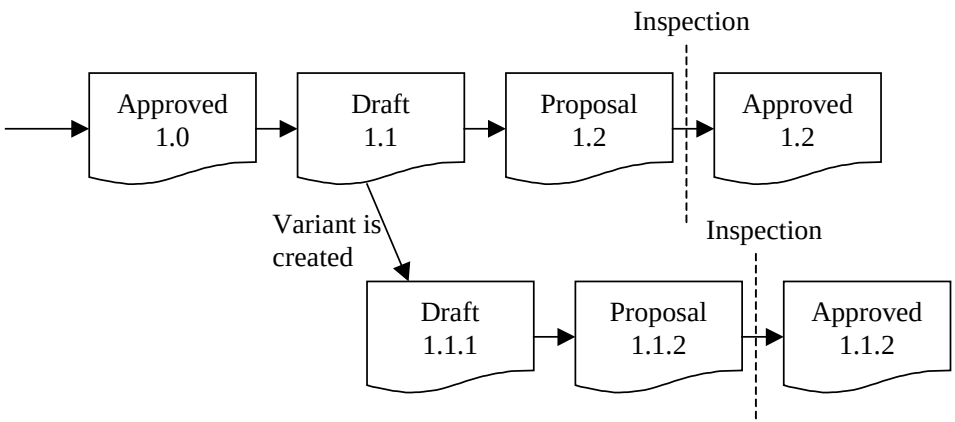


Figure 4. Versions and variants of a document.

2.2 Components and documentation

When comparing software systems, it can be seen that there are usually very high commonalities from one software application to another. This commonality includes code, design, and functional and architectural similarities (McClure 1994). Some of these similarities provide for reusable components.

A software component has several definitions in the literature. Generally, a component has been understood to be a self-contained, replaceable part of a software system that can be used independently or assembled with other components (Sametinger 1997). The component fulfils a clear function or a group of related functions through a well-defined and stable interface in the context of a well-defined software architecture. Thereby, components are units of composition with explicitly specified interfaces and quality attributes (Bosch 2000). A reusable component is developed (or acquired) for the solution of multiple problems (Karlsson 1996). A component model defines rules on how to construct an individual component and how the component interacts and communicates with other components (Heineman & Councill 2001). These design rules reduce interface mismatch and ensure that system-wide quality attributes are achieved. Thus, component models express the global (architectural) design constraints of components (Bachman et al. 2000).

SEI's (Software Engineering Institute) definition of a component is consistent with the overall thrust of these definitions, but however, it incorporates the perspectives of a component (Bachman et al. 2000): "A software component merges two distinct perspectives: component as an implementation and component as an architectural abstraction; components are therefore architectural implementations. Viewed as implementations, components can be deployed, and assembled into larger (sub)systems. Viewed as architectural abstractions, components express design rules that impose a standard coordination model on all components. These design rules take the form of a component model, or a set of standards and conventions to which components must conform."

Over the years, software systems have become more open. An open system is a collection of interacting components, whose interface specifications are fully defined and available to the public (Meyers & Oberndorf 2001). The openness allows the system be extended and integrated (Vigder 1998). In the last decade, the use of third-party components as parts of larger systems has grown considerably. Third-party components, such as COTS, MOTS, OCM and OS components, are developed by independent component providers.

Commercial Off-The-Shelf components (COTS) are bought from third-party vendors and then integrated into a system (Vidger & Dean 1997). They are

ready-made components with a certain functionality, and used without any internal modification by the customer. A COTS product is a product that is: "sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; supported and evolved by the vendor, who retains the intellectual property rights; available in multiple, identical copies; and used without modification of the internals" (Brownsword et al. 2000).

The use of COTS products and open systems promises a lot of benefits. According to Meyers and Oberndorf (2001), these include: "lower costs, less reliance on proprietary solutions, shorter development schedule, better-tested products, increased portability, increased interoperability and more stable technology insertion". In addition, the maintenance is carried out by the component vendor (Morisio & Sunderhaft 2000). However, COTS products also have some pitfalls: higher risk, inability to meet special requirements, conformance problems, support problems, increased amount of continual investment and requirement for a new management style (Meyers & Oberndorf 2001). In most cases, the main pitfall is the fact that the component buyer has no access to the source code (Vidger & Dean 1997).

A MOTS (Modified Off-The-Shelf) component is like a COTS but is tailored to buyer-specific requirements by the component provider. According to IEEE (1993), a MOTS product is "a product that is already developed and available, usable either as is or with modification, and provided by the supplier, acquirer, or a third party". A MOTS vendor needs to do careful coordination to determine how the modified versions will be maintained in the system and who will do that maintenance (Clapp & Taub 1998).

OCM (Original software Component Manufacturing) components locate between COTS components and components that are tailor-made. The term OCM describes the development of a tailored software component to be acquired by using intermediate component brokering services (Seppänen et al. 2001). OCM components are commercial components whose functional and quality requirements are defined by the integrator of the component, and the component is developed by a selected third-party component provider. Unlike COTS components, OCM components are supplied as a "white-box", which means that the components can be modified by the provider or the integrator before their actual reuse. The intellectual property rights become a serious

concern in the OCM business. The seller may have trouble supervising the component because the buyer is able to modify the source code of the component at any time (Seppänen et al. 2001).

Special attention has to be paid to OS components as well. Open Source (OS) components are different from proprietary components in several ways: programmers have unlimited access to the source code and components can be modified and redistributed without restrictions. The basic idea is that when all programmers can see the source code and exploit it, the software evolves, and the rapid evolutionary process produces better software than the traditional closed model. The OS license will automatically be applied to anyone using the component. Some considerations have to be taken into account when using OS components. Some license conditions may imply that the whole product in which OS components are integrated becomes software under the same license conditions as the integrated OS components. In addition, the organization has to plan the maintenance of the OS components. If a large organization modifies the OS components, it has to deliver the modified components back to the OS community; otherwise, the organization has to maintain the modified OS components itself. (Open Source Initiative 2002.)

Components, both commercial and in-house, do not have a standard documentation pattern and thus every component provider has documented components according to his own documentation practices. This means that documents can be inconsistent and of low quality. Some attempts have been made to develop a general component documentation model. However, these models are often appropriate only within a component reuse process inside an organisation, and these models do not consider the possible division of component development between different organisations. Further, some component documentation models for components offered in the electronic marketplace or repositories have been developed. These models do not usually require much information, and thus information that is crucial for component selection and integration may be missing or the information may not be specific enough.

Next, Reuse Manual (Sametinger 1997), the genre system of component documentation (Forsell & Päivärinta 2002), Component Registry (Component

Registry Homepage 2002) and COTStrader (Iribarne et al. 2002) documentation models are introduced.

2.2.1 Reuse manual

Sametinger (1997) classifies software documentation into parts according to a purpose: user documentation, system documentation, process documentation and reuse manual. User documentation provides all the information necessary for users to use the software system. Such information is, for example, a functional description of the system and an installation manual. System documentation provides the development information of a software system or component, such as requirements and implementation details. Process documentation describes the dynamic process of the creation of the component, such as the project plan and working papers. The last part, reuse manual, includes all the information concerning component selection, reuse and modification. The purpose of the reuse manual is to assure the software engineers that the component fits their needs and is therefore suitable for reuse in a certain scenario, and give information about the component's interfaces.

Figure 5 describes the structure of the reuse manual. The reuse manual consists of general information, reuse information, administrative information, evaluation information and other information (Sametinger 1997). The general information part provides the information needed when deciding if a component can be reused in a certain reuse scenario. This includes a short introduction to the component, the component classification, a description of its general functionality and platforms, and the status of the quality of the component. The reuse information part contains the information needed to install, reuse and adapt the component. This includes the installation instructions, a description of the interfaces and detailed information on how the component can be reused and adapted. The administrative information part includes the legal constraints, available support and history information. The evaluation information provides the more detailed information for the evaluation of the component. This includes the component's detailed functionality, information about the component's quality, limitations, interdependencies and test support. The other information part covers all the additional information that can be associated with the component, like system documentation and references.

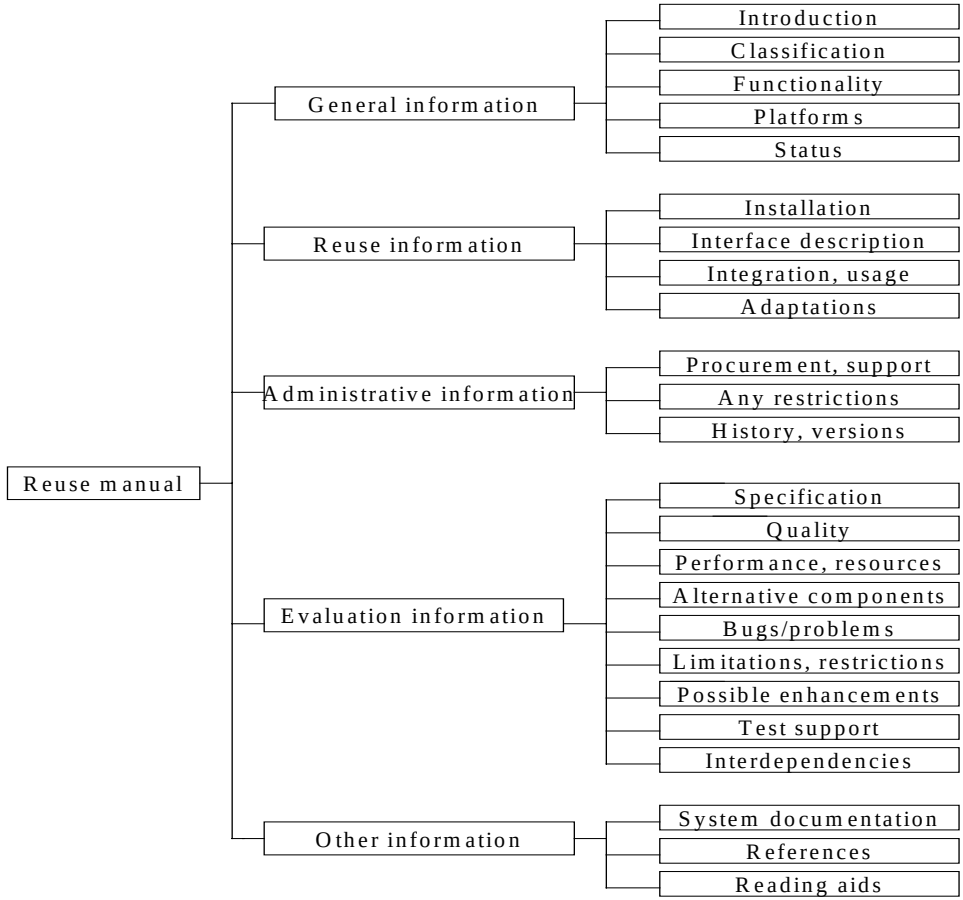


Figure 5. The structure of the reuse manual (Sametinger 1997).

2.2.2 Genre system of component documentation

Forsell and Päivärinta (2002) suggest an abstract-level genre system of component documentation that supports the reuse process. They define "genre" as an established way of communicating within a community that comprises a group of stakeholders involved in a component-based development process. They argue that the earlier models of component documentation are based only on the "pure expertise" of the writers. A documentation model that builds on a theoretical base is needed in the field of modern software societies. This kind of model takes notice of several dimensions of component documentation in practice:

- Component documentation is a continuously evolving communicative process of several stakeholders.
- Stakeholders have certain roles related to documentation, with their rights and responsibilities.
- With detailed structuring and standardisation of information content, the component offers a documentation framework and solid conceptual basis at the corporate level, and also at the industry level.
- Processing, sharing and storing structured documents denotes high technological challenges.

The communication actions related to the production and use of a component document are defined with four stakeholder roles: component provider, maintainer, software developer and reuse manager. The idea is to map the roles to all pieces of information produced during the for-reuse and with-reuse processes.

In genre systems, the documentation is divided into two parts: the reusable part and the part that supports reuse (Forsell & Päivärinta 2002). Figure 6 describes the structure of the information. The reusable part defines the objectives and design rationale for the component, results of the production work and the test procedures used to certify that the component works correctly. The part that supports reuse includes information for component brokering, consuming and management. The information content of the documentation is not defined precisely. The model does not bring any new information in comparison with previously developed documentation models.

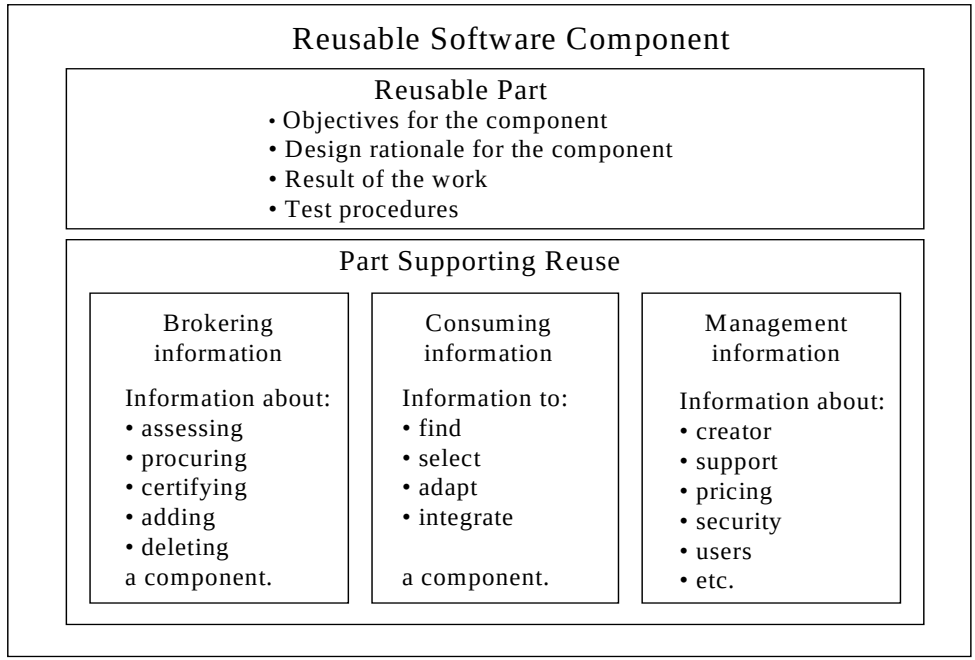


Figure 6. A model for documenting components (Forsell and Päivärinta 2002).

2.2.3 Component Registry

Component Registry is a fully searchable XML database of component documentation that covers a large collection of reusable JavaBean, Enterprise JavaBean and COM (Component Object Model) components (Component Registry Homepage 2002). XML (eXtensible Modelling Language) is a structured text standard that can be used to identify the parts of the document. Component Registry provides a universal organizational format for component documentation using XML formatting. The use of XML allows developers to repurpose the component data into any format and it also makes the component information portable and fully searchable.

Component Registry provides the Component Documentation DTD (Document Type Description) for documenting a component. DTD is a definition of the elements, attributes and entities of the document. The following information is required for each component in order to register the component into Component Registry's database (Component Registry Homepage 2002):

- component name,
- component version number,
- framework (Java or COM),
- category (and subcategory),
- a brief description of the functionality,
- technical documentation that may help a developer (code samples, FAQ, installation instructions, etc.),
- links to reviews about the component,
- technical support contact information, and
- a list of each class for Java components and a list of type library files for COM components.

Component Documentation DTD was placed in the Public Domain by Flashline Inc. in 1999. Flashline is one of the biggest electronic Java component marketplaces on the Internet. Components documented according to Component Documentation DTD can be searched from the database by browsing the list of components or by using the search option. The search retrieves the components from the database according to product name, vendor name, framework, technology, category or subcategory (Flashline Inc. 2002).

2.2.4 COTStrader

Most of the existing proposals for component documentation are based on the functional description and notion of component interfaces. The non-functional descriptions have usually been ignored. An effort has been made to include the non-functional information into COTS documents (Iribarne et al. 2001b). A COTS-XML Schema is a specification template for a COTS component that tends to cover all the documentation requirements. The template is based on the W3C's XML-Schema language (World Wide Web Consortium 2001b) and emphasises the importance of non-functional properties. Schema language describes the document content like a DTD, but Schema is more expressive, defining mechanisms for constraining document structure and content.

COTStrader is an Internet-based trader for COTS components. COTStrader provides XML schema templates for exporting COTS components into the COTStrader repository and importing COTS components from the repository (Iribarne et al. 2002). The current implementation of COTStrader is only a prototype but it is going to be extended in the near future.

The exporting template provides instructions on how to document a component. The template is divided into functional description, non-functional description, packaging or architectural description, and marketing description (Iribarne et al. 2002). The functional section defines the provided and required interfaces and service access protocols of the component. The non-functional section is used to describe the non-functional properties of the component, such as security and performance. Each non-functional property is described with a "property" element, with a type and value associated. The packaging section defines the packaging/architectural constraints of the component. Finally, the marketing section defines licences, certificates and other vendor information.

The importing template provides instructions on how to retrieve a component from the repository. Importing requires that the exact selection criteria of the component has to be known. For example, the property name and value must be declared before retrieving a component.

3. Software product lines

A software product line is a group of products that are developed from a common set of core assets but which differ among features that affect the buying behaviour of different customer groups (Clements & Northrop 2001). Core assets form the basis for the software product line. Core assets can be anything that can be reused in a domain, such as architecture, software components, domain models, requirements statements, specifications, performance models, test plans and process descriptions (Northrop 2002). A product-line approach is the development and application of reusable assets (technology base), the development of systems from a common architecture, and large-scale reuse of high-quality assets (NPLACE 2002).

The use of third-party components is increasing in product lines. Building systems as component-based product lines promises a lot of benefits to product development. The product developer is able to take advantage of new products and new technology, and the products' time-to-market is significantly reduced because Off-The-Shelf components are ready to use. The employees' productivity increases because the emphasis is not on coding but on reusing and integrating. The ready-made components allow developers to specialise in the application area of the organisation. The components are more reliable because the components used in a system were developed for a whole class of systems, and other users will probably have found more of the problems than if the component was built specifically for one system. The developed systems become more changeable because replacement components can be inserted in place of older, less satisfactory ones. The developed systems are also more extensible because new kinds of components can be inserted, allowing the system to perform some function it was not able to do before. (Bass et al. 1998.)

The product line architecture is a member of the collection of core assets. The architecture is expected to persist over the life of the product line and change relatively little, and relatively slowly, over time. Every product of the product line shares the common architecture. The architecture defines the set of software components and their supporting assets, such as documentation and test artefacts, that populate the core asset base. The architecture is used to create instance architectures for each new product according to its "attached" process (Northrop 2002).

In the following, the use of components in the product line context is examined. The essential characteristics of a component are defined from the perspective of the product line. These characteristics are then defined as component documentation requirements. Component development and selection for product lines are examined by defining the key roles in component development. Finally, the documentation requirements and their purpose on product line integrators are gathered.

3.1 Product development with components

In product lines, a new product is formed by taking applicable components from the base of common assets, tailoring them if needed and adding any new components that may be necessary. The components are assembled using a disciplined process under the umbrella of a common, product-line-wide architecture.

Figure 7 describes the product development process with attached elements. Several elements affect the process, such as requirements, product line scope, core assets and production plan (Northrop 2002).

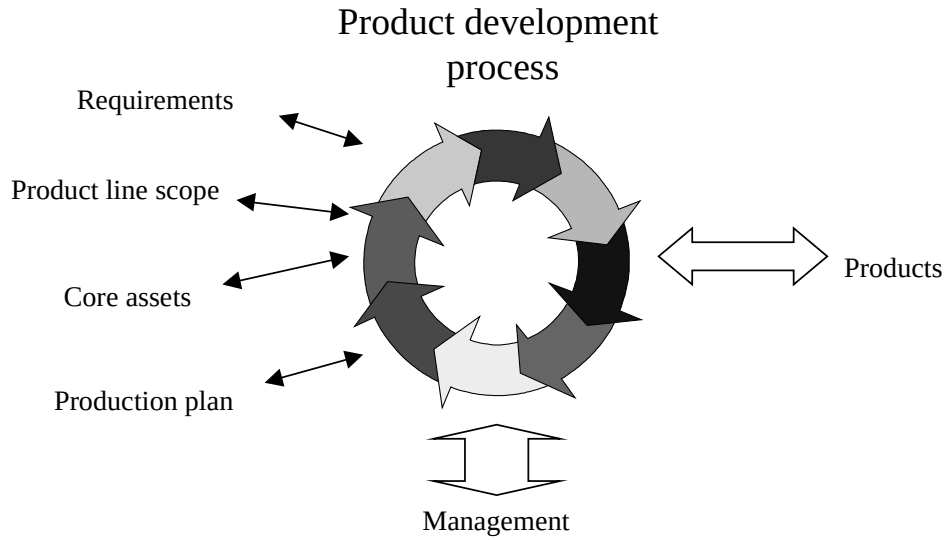


Figure 7. Product development in product line concept (Northrop 2002).

The product development process is guided by the requirements for a new product. In addition, the new product must obey the product line scope, which defines the commonality that all products share and the ways in which they vary from each other. The new product is built using core assets that are the building blocks of the products. Every core asset is not necessarily used in every product in the product line. Assets are chosen to achieve specific product features or attributes. The production plan defines how the core assets are to be used to build the products. Each of the core assets should have an "attached" process that prescribes its tailorability and use in the product development. Management gives resources, coordinates and supervises the product development activity (Clements & Northrop 2001).

The first step in the product development process is the requirements specification of a new product. This activity focuses on the functionality of the product that is not supported by the reusable assets of the product line. Typically, the requirements specification includes functional, performance, interface and design requirements, and development standards (IEEE 1990).

After the requirements specification, the architecture for the new product is derived from the product line architecture. Architecture pruning is an activity that removes the parts of the product line architecture that are unnecessary for the new product. Architecture extension activity extends the product architecture to cover the product-specific requirements that are not covered by the product line architecture. The extension can be implemented by extending the product line component or adding a product-specific component. Conflict resolution activity resolves conflicts, both functionality and quality attributes related, that may occur when deriving the product architecture from the product line architecture. In the end, the architecture assessment activity ensures that the architecture fulfils the product's requirements.

When the product architecture has been finalised, the next activity is to select a component implementation for each architectural component. Product line assets include component implementations that contain the functionality of the architectural components in the product line architecture. If no implementation for a certain architectural component is available, the component has to be developed or bought. Component implementation must often be instantiated to fit the product requirements. Component behaviour can be changed or adapted to

the product-specific context – for example, through the component's configuration interfaces.

Finally, the components are integrated as a product. The product is then validated through testing, packaged and released.

3.2 Product line requirements for components

The development or purchase of a new component is necessary in product development when a thorough search has been made of the existing product line assets and no applicable component is found. The choice of whether to purchase the component or to develop it is usually affected by the synergy of many different issues, such as the project's schedule, costs, resources, available components and reliable component suppliers (Meyers & Oberndorf 2001).

The product line places several restrictions and requirements on the new components. These are caused by the importance of the architecture for product lines and because the components possibly need to fit a potentially large family of systems. The resultant components can be product-specific components only used in a single product. New components can also be included in the core asset base of the product line when they can be used in multiple products.

The component requirements specification is the key to the component development or search. The requirements specification must include all the requirements and restrictions of the component that are prescribed by the product line architecture in which the component is intended to be used. The software architecture assumes that every component implements a certain required behaviour. Thus, a component requirements specification must define several aspects.

Bosch (2000) states that the interfaces, variability, functionality and quality attributes of the component must be carefully defined for every component of a product line. Thus, these properties should be included in the component requirements specification. To be able to assess the applicability of components for a product line, the same properties must also be defined for each candidate

component, so that the component's capabilities can be compared with the requirements specification.

The Software Engineering Institute (SEI) also identifies some characteristics for components that are peculiar to product lines. In addition to the Bosch (2000) definitions, these include such critical infrastructure elements as structuring architecture, architectural rules and implementation technique (Northrop 2002). In addition, SEI states that the resources that the component may require must be defined, as well as the protocols that the component has to adhere to and the component model that the component must support.

As stated above, several of the component’s properties have to be taken into account when developing or selecting a component for a product line. The defined, crucial properties of a component from the viewpoint of product lines are described in detail in the following section. Because the product line architecture has an essential impact on the component development or selection, some of the architectural issues of the software systems that integrate components are discussed in section 3.2.2.

3.2.1 The crucial properties of a component for product lines

When building systems by integrating components, the architectural issues become crucial. Every component is designed in a particular architecture and this architecture determines the assumptions concerning the interactions of the component with its environment. The importance of architecture is emphasised within product lines when the same product line architecture is expected to persist and be reused several times. The product line architecture effects on several component-related decisions. The architecture exposes the critical infrastructure elements that are common to all components of the architecture (Northrop 2002). Architectural rules and patterns also impose certain roles on the components.

When integrating a new component into a system, it must be verified that the component supports the style of the interactions of the system's architecture. This implies that the component interactions must be identifiable. Integration problems arise when the component depends on certain interaction assumptions

(Yakimovich et al. 1999). Figure 8 presents component interactions with its environment.

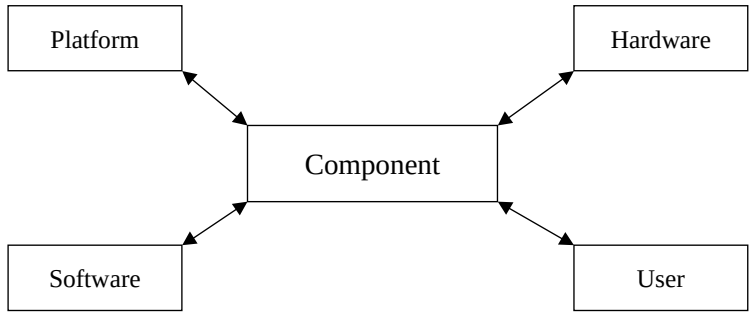


Figure 8. Interactions of a software component.

Component-platform interaction refers to the component's execution environment. If the platform does not match with the platform the component was designed for, it will need an emulator or a code converter to run the component. Component-user interaction dictates that component's user interface requirements may change and thus differ from the system user interface requirements. Component-hardware interactions indicates the direct interaction between the component and hardware. If a component's assumptions about the hardware are wrong, the component must undergo some modifications. Component-software interactions refers to other software components the system is composed of. (Yakimovich et al. 1999.)

- Furthermore, problems in component-software interactions include the following (Morisio & Sunderhaft 2000):
- functional – an internal problem of a component that is caused by a wrong or missing functionality,
- non-functional – an internal problem of a component; the component does not match with non-functional requirements,
- architectural – a class of problems that are caused by the architectural assumptions of a software component,
- conflicts – conflicts between the components in the system, and
- interface – incompatible interfaces between the components.

The functionality of the component specifies what the component actually does, i.e. what functionality it provides through its provided interfaces. The component obtains part of its functionality through its required interfaces but the greater part is implemented inside the component. Functional requirements for a COTS component need to be specified at a level that helps to evaluate the available components on the market.

Quality is the ability of the software or component to meet its requirements. The quality characteristics of the component are behavioural properties that the component must have and that should match the non-functional requirements (Kunda & Brooks 1999). The non-functional properties of the component must be defined so precisely that they can be measured. The common features of the non-functional requirements are that they are difficult to elicit, express, quantify and test (Beus-Dukic 2000).

Non-functional requirements can be specified with quality attributes. Bass et al. (1998) divide quality attributes into two categories: observable via execution and not observable via execution. Attributes observable via execution are defined as including performance, security, availability, reliability, functionality and usability; attributes not observable via execution are modifiability, maintainability, portability, reusability, integrability and testability (Bass et al. 1998). Sommerville (1992) groups the non-functional requirements into product requirements, process requirements and external requirements. Product requirements, such as portability, usability and reliability, are placed on a system under development. Process requirements are placed on the development process used for system or component development. These can include process standards, implementation requirements or design methods. External requirements are other requirements placed on a component, e.g. interoperability of the component and legislation requirements.

Bertoa and Vallecillo (2002) have made a proposal of the quality attributes that measures the characteristics of COTS components. It uses the ISO 9126 quality model of a software (ISO/IEC-9126 1991) as a basis, but modifies it, claiming that not all the characteristics of a software product defined by it are applicable to COTS components. They divide quality attributes into two main categories: attributes measurable at run-time and attributes observable during the product's life cycle. The first category includes accuracy, security, recoverability, time

behaviour and resource behaviour. The second category includes suitability, interoperability, compliance, maturity, learnability, understandability, operability, changeability, testability and replaceability (Bertoa & Vallecillo 2002).

Functional and non-functional problems cannot be solved without reworking or modifying the component itself. The rest of the interaction problems can be solved within the limits of the software architecture by adapting components. As stated above, the software architecture defines the relationship that the component has with other components. Each relationship defines an interface that is, according to Bosch (2000), "A contract between a component requiring a certain functionality and a component proving that functionality. The interface represents a first-class specification of the functionality that should be accessible through it. The interface specification is, ideally, independent of the component or components implementing that interface."

Component interfaces can be divided into required, provided and configuration interfaces. A component interacts with other components through provided and required interfaces. Interfaces are usually represented in the form of application programming interfaces (API) – that is, a specification of those properties of a component that component integrators can depend upon (Bachman et al. 2000). API specification contains a list of operations, generally including argument types and the type of entity returned (Clements & Northrop 2001). Some particular aspect should be taken into account when specifying interfaces. The state of the component affects the interface and, therefore, all interface operations are not accessible at all times. In addition, multi-component interaction assumes some kind of protocol to dictate the communication between the components (Yakimovich et al. 1999). In special cases, a set of abstract classes that are visible to users of the component can be part of the interface, and should be defined in association with the interfaces (Clements & Northrop 2001). The third interface, the configuration interface, provides a point of access for the component user to configure the component instance according to the requirements. This interface allows the user to set parameters that select particular variants at the variation points (Bosch 2000).

An interface specification describes how individual services respond when invoked. As components are integrated, additional information about component communication is needed. A protocol groups together a set of messages from

both communicating components and specifies the order in which they are to occur (Northrop 2002). The resource that the component needs should be documented, so that the possible conflicts between the resources of the application and the needed resources of the component can be detected (Clements & Northrop 2001).

To be applicable in more than one context, the behaviour of the component must be variable. Variability occurs at different levels in the design: product line level, product level, component level, sub-component level and code level (Svahnberg & Bosch 2000). Component variability refers to the points where the behaviour of the component can be changed (Bosch 2000; Northrop 2002). These points for component users are provided by configuration interfaces. Thus, at the component level, the variability consists of how to add new implementations of the component interface (Svahnberg & Bosch 2000). Components that are included in the core asset base must support the flexibility needed to satisfy the variation points specified in the product line architecture, and/or the product line requirements. The architecture also defines those places at which variation is allowed.

Svahnberg and Bosch (2000) describe five variability mechanisms: inheritance, extensions, configuration, template instantiation, and generation. These mechanisms provide the appropriate functionality or setting at a variation point of a component. Configuration through parameter settings or templates is the most typical usage (Bosch 2000). The configuration variability mechanism handles variability by including all variants at all variation points in the component and providing an interface to the component user, whereas the template instantiation variability mechanism allows components to be configured with application-specific types.

Component implementation technologies determine component-related features, such as the binding times that are available and the architecture of the run-time environment. The product line architecture will include a basic component model as the foundation for the implementation. The component model sets the requirements and restrictions for the component implementation and communication.

3.2.2 Architectural mismatch

Components developed by third-party organisations may not meet all the requirements of the component integrator, or they may not even work with other product line components. Special attention has to be paid to the system architecture when integrating COTS components. The system architecture must allow the substitution of components, so that a component can be replaced without any influence on the system. The architecture must also allow the isolation of components, so that there can be only minimal coupling between components. COTS components are often overloaded with functionality, and the architecture must provide a mechanism for hiding the unwanted functionality of a component. The architecture must also adapt to the connectors and functionality available in the components and possible different data models and data formats used by the components. (Vidger & Dean 1997.)

An architectural mismatch occurs when the integrated component has defective assumptions about the system. Four main categories of architectural mismatch in component integration have been identified. These categories are based on assumptions the components make about the structure of the system and its environment (Garlan et al. 1995):

1. The nature of the components

- Infrastructure. Assumptions about the substrate on which the component is built.
- Control model. Assumptions about which components (if any) control the sequencing of computations overall.
- Data model. Assumptions about the way in which the environment will manipulate data managed by a component.

2. The nature of the connectors

- Protocols. Assumptions about the pattern of interaction characterised by a connector.
- Data model. Assumptions about the kind of data that is communicated.

3. The global architectural structure

- Assumptions about the typology of the system communications.
- Assumptions about the presence or absence of particular components and connectors.

4. The construction process

- Assumptions about the order in which pieces are instantiated and combined in an overall system.

In order to minimize the architectural mismatch, the COTS components have to be adapted to the system architecture. This adaptation can be realized by using integration techniques. The main technique is the use of integration components that are black-box techniques applied without access to the COTS component source code. The integration components can be defined to include wrappers, glue and tailoring (Vidger & Dean 1997). A wrapper is a piece of code that is built to isolate the COTS component from other system components. Glue is a code that provides the functionality to combine the different components. The glue code's functionality should not depend on a specific component but it should allow the substitution of components. Tailoring means adding some element to the component to provide it with additional functionality. COTS components should always have glue or wrappers between them, because integration code is the only source code to which the developer has access and by which the developer can control the interactions of the components (Vidger & Dean 1997).

3.3 Component development and selection for product lines

The common design phases of software, such as analysis, design, implementation and testing, include certain tasks that are fulfilled by stakeholders – i.e. persons involved in software development. Each stakeholder plays certain roles in the software development. Every role is responsible for a set of activities, such as delimiting the system, designing the use cases, coding, planning or doing the testing. The roles in the software development can be defined to include, for example, a system analyst, a use-case specifier, an

architect, a component engineer, a system integrator and testers (Jacobson et al. 1998).

In the same way, the essential roles can also be defined in component development. The idea of defining the roles in component development has been in the separation of the for-reuse process and the with-reuse process. For-reuse signifies the systematic development of reusable components. Development for-reuse requires that the variability in requirements between different reusers has to be analysed and the component has to be designed with the appropriate level of generality for all reusers (Karlsson 1996). With-reuse can be defined as the systematic reuse of components as building blocks in creating new systems. Development with reuse includes activities of searching for a set of candidate components, evaluating them to find the most suitable and, if necessary, adapting the selected component to fit the specific requirements (SER Consortium 1996).

COTS and OS components are developed in the for-reuse process on the component provider's side. Due to simplicity, the stakeholders involved in the COTS and OS component development can be defined to include a component architect, a component designer and a component developer (Niemelä et al. 2002). The component architect defines the functional and quality properties that the component has to satisfy and validates the developed component. The component designer is responsible for designing the details of the component according to these requirements. Component developers are those who implement and test the component.

The with-reuse process on the component integrator's side denotes that the integrator builds products by integrating components. The integrator acquires COTS or OS components with a special acquisition process or orders OCM components. The integrator specifies the requirements for the required component and then searches or orders the applicable component. The component integrator's roles include a product line architect, a reuse manager, a product architect and a product developer (Niemelä et al. 2002). The reuse manager is a person or persons responsible for the repository of reusable assets, i.e. the selection, validation and maintenance of components. The product line architect ensures that the selected components fulfil the requirements of the product line, such as quality requirements and component communication

standards. The product architect designs the new product, which includes this new component. The product developer assembles and tests the new product.

In the case of an OCM component, the for-reuse process is split between two organisations: the component integrator and the component provider (Kallio & Niemelä 2001). Requirements are defined on the integrator's side by the product line architect and the reuse manager, and the rest of the component is complemented on the provider's side. The two organisations communicate directly with each other.

Figure 9 describes how the for-reuse and with-reuse processes are related in component development and acquisition.

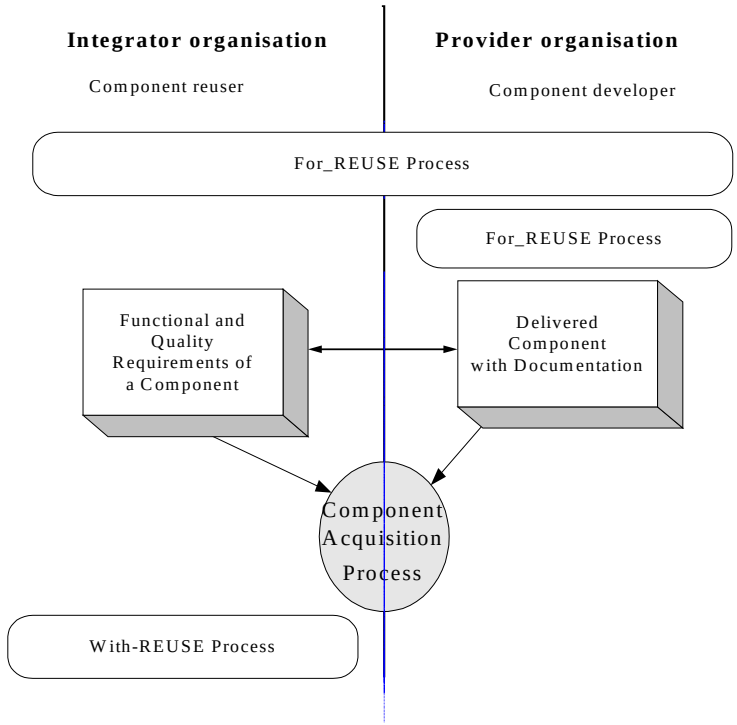


Figure 9. Component development and acquisition (derived from Kallio & Niemelä 2001).

The component integrator orders OCM components directly from a selected component manufacturer. COTS components are also acquired through a well-defined acquisition process. The acquisition process usually includes several activities, such as COTS cost estimation, vendor qualification, risk management, market research, candidate COTS evaluation, and suitable COTS selection and procurement (Meyers & Oberndorf 2001).

According to Meyers & Oberndorf (2001), the selection of the component is based on vendor qualification and the COTS evaluation results. Component evaluation is done by acquiring information about the component and assessing how well the component meets the evaluation criteria. Therefore, in component acquisition, the component documentation has a great emphasis in the component evaluation and thus on component selection.

The selection of a suitable COTS component requires a well-defined selection process. A repeatable process makes the planning easier, increases the efficiency and enables learning from previous cases (Kontio 1996). The selection process has three phases: criteria definition, identification of candidate components and evaluation (Kunda & Brooks 1999). The criteria definition phase defines the requirements for the component and converts them into evaluation criteria sets. These criteria sets are mainly component functionality, required quality characteristics, business concerns such as vendor guarantees and legal issues, and software architecture constraints. The identification of candidate components includes searching the alternative components that can be assessed in the evaluation phase. In the evaluation phase the properties of the components are identified and assessed according to the evaluation criteria, and the most suitable component is selected.

The selection of a COTS component is often a problematic task and sometimes it is impossible to find a component that satisfies all the requirements. Kunda & Brooks (1999) state that the black-box nature of COTS components is one of the major problems in COTS evaluation and selection. The lack of access to the component's internal parts and insufficient supporting information makes COTS understanding and evaluation hard.

In many cases, the selection of a component is a compromise among user requirements, system architecture and a COTS product (Iribarne et al. 2001a).

Once two or more candidate COTS components that meet the core functional requirements have been identified, the non-functional requirements are used in the selection process between the candidates. Thus, if two components have the same functionality, a careful analysis of the quality attributes can be used as decisive criteria.

Several attempts have been made at making the selection of a component easier. PORE (Procurement-Oriented Requirements Engineering) is a COTS selection method that guides the acquisition of customer requirements and the selection of a COTS software or component that satisfies these requirements (Ncube & Maiden 2000). The information used in the selection is based mainly on simple software component and supplier information, supplier demonstrations, hands-on COTS component use and emergent system properties from user trials. OTSO (Off-The-Shelf Option) is a COTS selection method that is systematic, repeatable and requirement-driven (Kontio 1996). The criteria set is specific to each COTS selection case, but, in the main, the criteria can be categorised into four groups: functional requirements for the COTS, required quality characteristics, business concerns, and relevant software architecture.

The National Product Line Asset CEnter tends to produce decision-making information for selecting software products, especially for product lines (NPLACE 2002). NPLACE is an independent software testing facility sponsored by the Air Force Electronic Systems Center. NPLACE has developed a set of criteria that are COTS-product specific. These criteria help to select a COTS component for any product line. A hierarchical model for certifying COTS components for product lines has been developed from these criteria (Yacoub et al. 2000).

A hierarchical reference model for COTS certification criteria has four levels: COTS-Worthiness, Domain Pervasiveness, Architecture Conformance and Application Adequacy (Yacoub et al. 2000). Each of these levels considers different aspects of the component when trying to define if the component is suitable for a product line. The criteria are defined in Figure 10. At the COTS-worthiness certification level the goal of the certification is to ensure that the component is technically and commercially sound. The component has to meet criteria that deal exclusively with its content. At the domain pervasiveness level the goal of the certification is the usefulness of the component in the domain.

Architecture conformance criteria assess the usability of the COTS component in a given product line context. Application-specific criteria assess the adequacy of the component for one particular product of the product line.

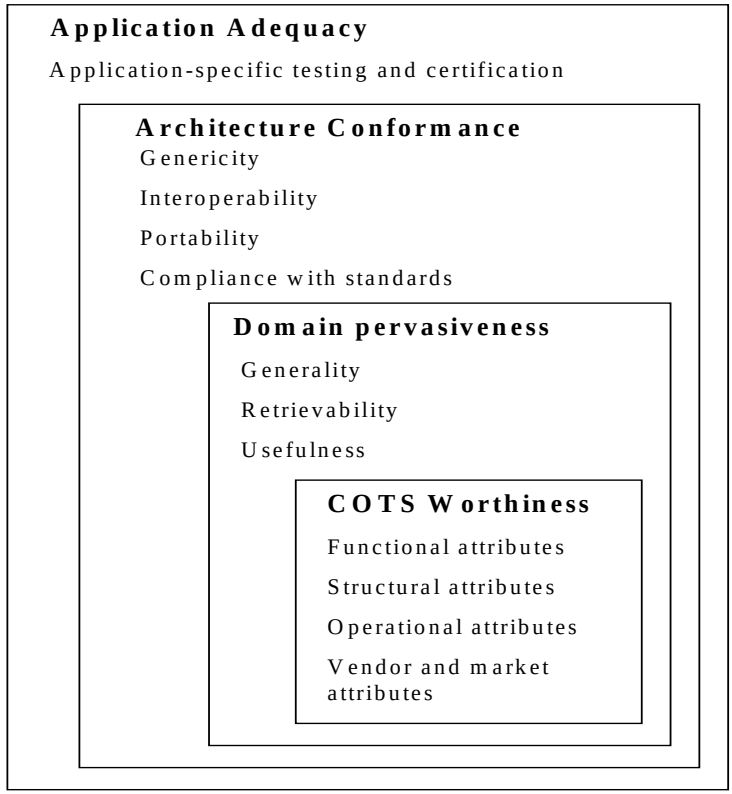


Figure 10. COTS Certification Hierarchy (Yacoub et al. 2000).

To be able to assess components on the basis of these criteria, the information for each criterion should be defined by a component provider. Thus these criteria can also be thought of as extensive requirements for component documentation.

3.4 Component documentation requirements

The essential characteristics of components from the viewpoint of the product line were discussed in section 3.2.1. The component documentation should provide information that assists the component integrator to assess whether the

component fulfils his product line architecture's requirements and restrictions for a component or not. The component integrator must be able compare his requirements for a component with the documentation of a candidate component, and then assess if the component is applicable for his purposes.

The component documentation should thus include information that the product line defined as essential. Furthermore, some component documentation requirements can be derived from the previous documentation models discussed in section 2.2 and from the evaluation criteria specified in the hierarchical reference model for COTS certification (Yacoub et al. 2000) presented in Figure 10.

Component documentation has several purposes for the component integrator (the buyer). It assists in:

- searching for components,
- selection of components,
- validation of interoperability of components,
- analysis of combined quality attributes,
- integration of components,
- use of components, and
- maintenance of components.

The component search can be helped by the document structure, keywords and implementation technique. The required information is easy to find from the document when structuring documents so that the information follows the defined hierarchy. In addition, structured document implementation techniques enable the retrieval of possible candidate components based on, for example, document elements or attributes. Keywords added to documents enable the retrieval of components using search words.

The selection of components can be assisted by providing the information needed when assessing the applicability of a component to the requirements. This can include, for example, the description of the basic functionality of the component, interfaces, execution environment and constraints. Also, the

information about the quality of the component, the component implementation technique, the architectural rules and the variability of the component should be available because this kind of information is critical, especially when choosing a component for the product line. Other component-related information, such as the price of the component, the reputation of the component supplier and supply contracts, affect the component buying decision but these should not be included in the component documentation because they are not product-specific properties and change over time.

By providing information about the protocols, component model and use of the interfaces, the documentation can help the integrator to verify the interoperability of the component with other components, both commercial and in-house components. In the case of product lines, the interoperability rules are defined by the product line architecture, and, therefore, it is extremely important to describe how the component communicates with other components.

The documented quality attributes often provide information about the adaptability, portability and performance of the component. The quality should be defined in a measurable way and the validation of the quality attributes must come out in a document, e.g. in association with acceptance testing. Both the quality and functionality of a component can be validated for integrators by providing information about component acceptance.

The documentation should provide information for the integration of the component with other components of the system. Detailed interface descriptions and possible examples of the use of the interfaces enable the integration of the component. Also, the interdependencies, both to other components and physical resources, must be known.

To be able to use the component, the documentation must provide information about the installation, configuration and tailoring of the component. Installation information guides the user in getting started with the component. Tailoring information helps the user to fit the component to his own needs. Configuration information specifies the component variation points and how to use them.

The maintenance information helps the user to maintain the component. The design models of a component, such as class diagrams or context diagrams, help

the user to modify or extend the component. The history and delivery information is needed for component updates and storing. The document should also contain a point of contact for a user to get help with the use and maintenance of the component.

Table 2 summarises the documentation requirements defined above. The requirements are discussed more details in the next chapter, in association with the developed documentation pattern.

Table 2. The derived component documentation requirements.

The use of documentation	**Documentation requirement**
Component search	Keywords
Component selection	Functionality Quality attributes Interfaces Constraints (inc. standards and protocols) Architectural rules Technical details (i.e. execution environment)
Component validation	Test criteria Acceptance testing
Analysis of the component quality	Quality attributes Acceptance testing
Integration of a component	Interfaces (detailed)
Component use	Installation guide Configuration Implementation Resource needs Interdependencies
Component maintenance	Component identification History Composition Implementation Delivery Tailoring support Customer support

4. A component documentation pattern

The lack of standard, self-contained documentation of software components is the main reason for component selection and integration being hard, especially in product lines where the functional and technical requirements, quality, constraints and design rules for a component are specified within product line architecture. This chapter provides a general pattern for documenting software components in a standard way whereby the component integrator's and provider's views are taken into account. COTS components are documented by the component provider only, but in the case of OCM components the component integrator and the provider have to co-operate and follow standard documentation procedures to guarantee that the delivered component meets the requirements.

The developed documentation pattern emphasises the documentation requirements derived from the literature, including the product line requirements for the components. The pattern defines the required information and structure for the component documentation. The pattern follows the general design phases of component development and thus enables rational creation of documentation for components.

4.1 Content of component documentation

The information on the component documentation is derived from the literature (Sametinger 1997; Wallace et al. 1992; Heineman & Council 2001) and, furthermore, refined by the documentation requirements of the software components for product lines (Bosch 2000; Northrop 2002). Figure 11 describes the simplified structure of the documentation pattern. The information is divided as follows (Niemelä et al. 2002):

- **Basic information** defines the identification and properties of a component. It describes the component's general information, and responsibilities of the component from the architectural point of view. These descriptions are described before developing or purchasing a component and thus respond to the component requirements specification.

- **Detailed information** provides more detailed information about the functional and quality properties, as well as the technical details. The description of the component's design and implementation is given. For a COTS component this implementation description is usually restricted because of the COTS component's black-box nature.
- **Acceptance information** guides component selection and validation, and proofs the quality of the component. The information corresponds to the component acceptance test.
- **Support information** helps component users to maintain a software component. It provides a point of contact for finding help in problem situations and in the adaptation of a component.

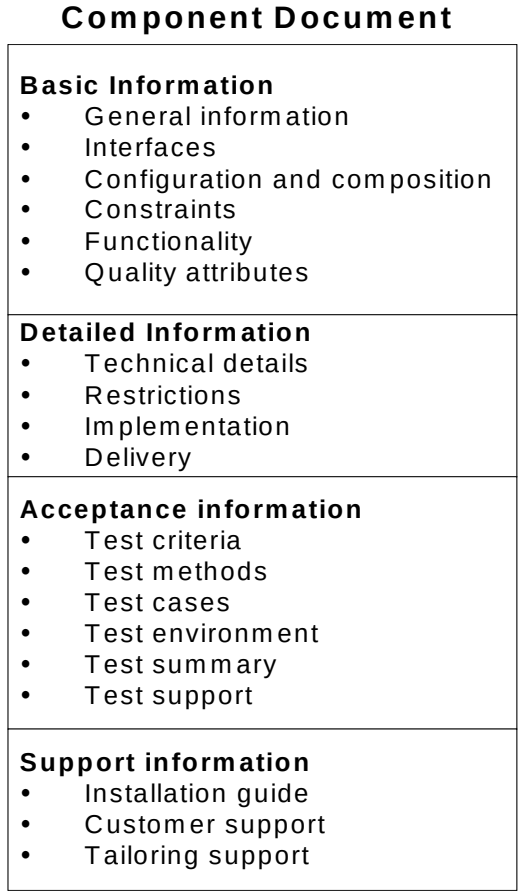


Figure 11. The main structure of the component documentation pattern.

In the following, the pattern is described more precisely. The information is explicated with the documentation of an example component. The example component is an architectural-level OCM component that is developed solely as an example. Thus the documentation of the example component is not complete.

All the required information introduced in the next sections may not be available for every component. For large and complicated components, most of the information of the documentation pattern can be identified, unlike small and simple components. In addition, some of the information, such as modifiability and some sections of the implementation, may only be adequate for OCM components.

4.1.1 Basic information

The main structure of the basic information is presented in Figure 12. The basic information has six parts: general information, interfaces, configuration and composition, constraints, functionality, and quality attributes. The general information of a component gives an overview of a component with the following pieces of information: component identification, type, overview, history, and special terms and rules. The interface information defines both the required and provided interfaces of the component. Configuration and composition concentrates on the form of altering the component. Constraints defines the protocols and standards the component has to apply. Functionality provides a detailed functional specification. Because the quality has a great emphasis in the software architecture, the quality of the component must also be identifiable.

1 BASIC INFORMATION
- 1.1 General information
  - 1.1.1 Identification
  - 1.1.2 Type
  - 1.1.3 Overview
  - 1.1.4 History
  - 1.1.5 Special terms and rules
- 1.2 Interfaces
  - 1.2.1Version usage
  - 1.2.2 Required interfaces
  - 1.2.3 Provided interfaces
- 1.3 Configuration and composition
- 1.4 Constraints
  - 1.4.1 Protocols
  - 1.4.2 Standards
- 1.5 Functionality
- 1.6 Quality
  - 1.6.1 Modifiability
  - 1.6.2 Expandability
  - 1.6.3 Performance
  - 1.6.4 Security
  - 1.6.5 Reliability

Figure 12. The structure of the basic information.

General information

Component identification separates the component from other components. The identification includes the component number and name. At least the identification number must be unique. The component name should be well defined and describe the component. Component type expresses the way the component is intended to be used. The component type can be, for example, a subsystem, an actor component, a function block or a procedure. Because the documentation may be long, it is useful to provide a short comprehensive description of a component that can be browsed quickly to obtain a general picture of a component. The overview section provides this brief description – i.e. what the component does. It describes the design rationale, applicability and objectives of the component. The history describes the life cycle of the component. This includes the component version and revision date, the persons who have made the work, and the initial version and modifications that have been made to the component. The component's special terms and design

rationale must also be defined. The design rationale may define the architectural choices – for example, the architectural styles and patterns the component has to follow, and the reasons behind the design decisions (Matinlassi et al. 2002). OCM components should follow the same terms and design rationale as the product line architecture in which it is intended to be used. The general information of an example component is described in Table 3.

Table 3. An example application of a component's basic information.

1.1 General information	
1.1.1 Identification	
• Id	7986764sd
• Name	LocationServiceManager
1.1.2 Type	Service component
1.1.3 Overview	This component provides a location service. It receives, delivers and maintains dynamic location information of different services and data objects in the distributed system. It acts as a central location information server. It enables various services and software components to register and unregister their location information.
1.1.4 History	
• Version	1.0
• Date	June 2001
• Developer	Mauri Matinlassi, VTT
• Improvements	This component does not have previous versions.
1.1.5 Special terms and rules	
• Term	Location origin
• Description	Contains the location information of a service (e.g. in which device the service locates, how to contact the service)
• Rule	Communication rules
• Description	Client-Server architecture style, asynchronic communication

Interfaces

The interface information is derived from the structural view of the product line architecture (Matinlassi et al. 2002). Interfaces determine how a component can be used and interconnected with other components (Sametinger 1997). Table 4 shows how the pattern is applied to the required and provided interfaces of the case example.

Table 4. The interface descriptions of the example component.

1.2 Interfaces

1.2.1 Version usage — N/A

1.2.2 Required interfaces

Interface

- Name — DataStorage interface
- Type — Basic messaging interface between two services.
- Description — This interface provides communication with Distributed Data Storage (DDS). Location information is saved into and updated in the DDS component data structures through this interface.
- Behaviour — Static interface
- Interface functions

  Function
  - Name — saveLocationOrigin
  - Description — Stores a location origin to Distributed Data Storage.

  Function
  - Name — updateLocationOrigin
  - Description — Updates the data of Location Origin according to changes.

1.2.3 Provided interfaces

Interface

- Name — Serving interface
- Type — Basic communicating interface between middleware services.
- Description — Mobile services, e.g. Lease Service and Directory Service, register, unregister and update their location information to LocationServiceManager through this interface. Directory service provides a directory service for user applications. Lease Service provides lease services for user applications and the directory service. These mobile services communicate with Location-ServiceManager in order to give their location to other members in the network.
- Behaviour — Static interface
- Interface functions

  Function
  - Name — addObserver
  - Description — This signal refers to a method addObserver (Observer o) in public class java.util.Observable. Adds an observer to the set of observers for this object. Here it adds a service as an observer of LocationServiceManager in order to get update notifications when business card information has changed.

  Function
  - Name — getBusinessCard
  - Description — Creates a new business card and sends it to the service who sent the request.

Both the required interfaces and the provided interfaces of a component have to be defined. The required interfaces are those which the component needs to operate, and the provided interfaces are those which the component offers to other components. The interface information is here defined as including the interface name, type, description, behaviour and interface functions. The main interface definition is given in the description part. The behaviour part describes the behaviour of the interface. Interface functions describes the activities of the interface. Version usage tells what version of the defined interface is in use and what the practice in the version management.

Configuration and composition

Configuration and composition describes how the behaviour of the component can be changed and how the component is included in a software system. Configuration describes a way of initiating the component in different contexts – i.e. to vary the component. Table 5 describes the configuration and composition of the component.

In internal composition, the component is included in a software system – for example, by linking binary code while assembling or including source code in a system. In external composition, the component runs as an independent program. Functional and object-oriented compositions are based on the activation of the components by function calls (Sametinger 1997). However, in object-oriented composition different components are combined through polymorphism and dynamic binding. Textual composition is used for macros and parameterisable components. Macros can represent reusable components that are modified according to parameters and inserted at the location of their reuse. Configuration and composition is especially important in software product lines because it defines the variation points and variants provided at the component level.

Table 5. An example of description of configuration and composition.

1.3 Configuration and composition	Internal composition, included in a software system as a source code.

Constraints

Constraints of components are usually common constraint for all the components in a software system, both made in-house and third-party components. In the case of an OCM component, the constraints are defined at the product line architecture level. Constraints are here defined as including both protocols and standards.

Service Access Protocols (SAP) define the global behaviour of a component. Table 6 specifies the protocols of the example component. A protocol describes the interaction between two components needed to achieve a specific objective. It specifies in which order the methods of the two interacting components has to occur. Protocol information can be expressed in many notations, such as Petri Nets, Message Sequence Charts, etc. (Iribarne et al. 2001b). Here, ServiceAccessProtocols information includes the protocol name and description.

Standards can restrict the compatibility, structure and functionality of components. The standards of the example component are described in Table 6. The component model defines the architecture of the component. It also determines the global behaviour, how the component interacts with other components in a system (Heineman & Council 2001). Three basic models are currently the most popular: Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM) and Java/Remote Method Invocation (Java/RMI) (Morisio & Sunderhaft 2000). In addition, both required and used standards should be specified.

Table 6. The constraints of the example component.

1.4 Constraints	
1.4.1 ServiceAccesProtocols	
Protocol	
• Name	Protocol for location information registering
• Description	Service sends a request for a LocationOrigin. LocationOrigin is created and stored, ID sent as a response. LocationServiceManager adds itself as an observer of a service that requested LocationOrigin. LocationServiceManager is notified if the service has changed its location or passed away.
Protocol	
• Name	Protocol for location information requesting
• Description	Service sends a request to contact a service. Something is known (e.g. name) but not the location information of that service. LocationServiceManager creates and sends a Business Card. Service and adds itself as an observer of LocationServiceManager in order to get update notification of business card changes.
1.4.2 Standards	
Used standards	
• Component model	N/A
• Other standards	HTTP, SMTP
Required standards	IPv6

Functionality

Functional specification is derived from the behaviour view and the deployment view of the product line architecture (Matinlassi et al. 2002). Functionality describes the supported functions of the component. For every function the function name, description, inputs and outputs are specified. Function description declares what the component does - i.e. the description of all operations, equations, mathematical algorithms, logical operations, etc (Wallace et al. 1992). The required data is also described, together with functional exceptions if the architecture requires their being handled in a specific way. Table 7 specifies the functional description of the case example.

Table 7. The functional description of the example component.

1.5 Functionality	
Function	
• Name	Create Location Origin
• Description	Location origins for mobile services are created on request.
• Inputs	createLocationOrigin request
• Outputs	Location Origin capsule is plugged in addObserver message sent to the owner of the new origin
Exceptions in functionality	N/A

Quality

Quality in this context means the quality attributes that the component embodies. In the case of an OCM component, the quality of the component is defined by the product line architecture. The most important quality attributes for components are here defined as including modifiability, expandability, performance, security and reliability. Table 8 provides a description of the quality of the example component.

Modifiability defines how the component can be modified to a new environment and how it can be adapted to new changes quickly and cost effectively (Bass et al. 1998). COTS components are often black-box components, and this attribute is often qualified only for OCM components. Expandability describes how new features can be added to the component and thus expand the functionality of the component. Performance is a quality attribute that measures the component. The measurements are the size of the component, prioritisation of events, capacity, throughput, error detection and allocation time of resources (Wallace et al. 1992). The size of the component can be expressed, for example, by the lines of code (Park 1992). Prioritisation of events defines any events with a higher priority that must be handle first. Capacity measures the amount of work a component can perform. Throughput measures how many events or how much data the component can handle in a given time unit. Error detection describes how the component copes with error situations. Allocation time of resources defines the time unit the component allocates to a certain physical resource. Metrics that measure performance can be collected during the development of the component (priori metrics) and when a component is used (posteriori

metrics) (Karlsson 1996). Security is a quality attribute that defines the strategies against viruses and hackers. It also describes recovery methods in attack situations and methods for protecting the data. Reliability defines the time that the component operates or its ability to perform its required functions under stated conditions (ISO/IEC-9126 1991).

Table 8. The description of quality of the example component.

1.6 Quality	
1.6.1 Modifiability	Loosely coupled with other components because of the publish-subscribe architectural style. The number of mobile services in a system can be extended without modifications to LocationServiceManager. The number of components requesting services from Location-ServiceManager can be extended without modifications to LocationServiceManager. Internal functionality is encapsulated and can be updated or modified without any change to other components.
1.6.2 Expandability	N/A
1.6.3 Performance	
• Size	200 lines of code.
• Prioritization of events	All the service requests have equal prioritization. LocationServiceManager responds to service requests in order of arrival.
• Capacity	LocationServiceManager can handle the location information of up to 500 service components.
• Throughput	LocationServiceManager is capable of handling at least 48 requests every second it is up and running.
• Error detection	When sending messages to a dead service, an exception is caught.
• Allocation time of resources	0.03 ms processor execution time per every service request.
1.6.4 Security	N/A
1.6.5 Reliability	Location origin data is saved in several replicas in order to guarantee the safety of information.

4.1.2 Detailed information

Figure 13 presents the structure of the detailed information. The detailed information includes details of the component design and implementation. The detailed information has four parts: technical details, restrictions, implemen-

tation and delivery. Technical details includes the application area, development environment, platforms, interdependencies, prerequisites and special physical resource needs. Restrictions concentrate on design and implementation restrictions. The implementation part describes the implementation models of the component. The delivery part provides information about component's delivery.

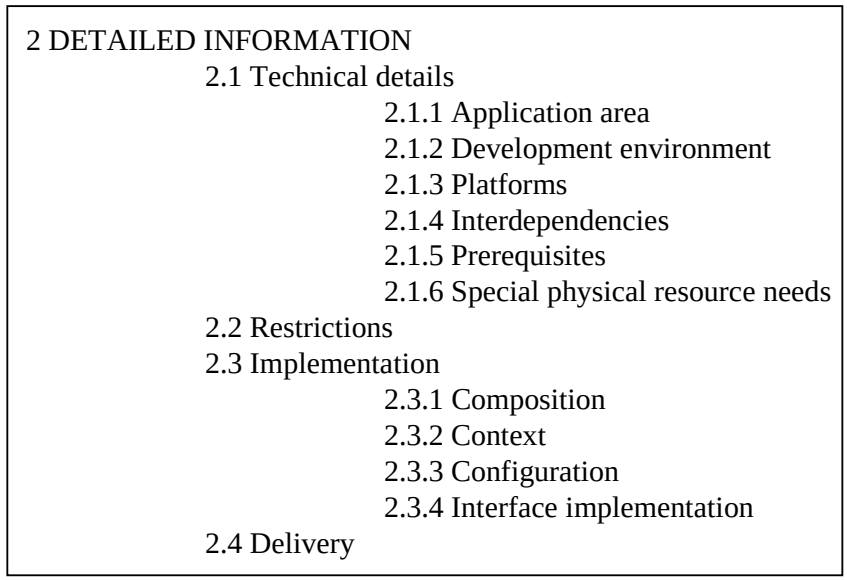


Figure 13. The structure of the detailed information.

Technical details

The application area defines the areas the component is developed to or can be used in. Development environment defines the environment in which the component has been developed. Typical examples of such environments are Java Virtual Machine, Jini and MicroSoft Foundation Classes. Platform describes any hardware and software the component is built on. Platform also indicates the reusability of the component. The component is more reusable the fewer platforms it depends on. The platform can be an operating system, a set of libraries, a compiler, etc. (Sametinger 1997). For every platform, both software and hardware, the platform name and description must be specified.

Interdependencies describes component's dependency on other components. The required components have to be named and the reason for the dependency has to be specified. Prerequisites defines all the other requirements that component may have to operate. These can include, for example, Java classes or type libraries associated with the component. Special physical resource needs declare the physical resources that the component requires to operate. Table 9 presents an example description of the technical details.

Table 9. Technical details of the example component.

2.1 Technical details	
2.1.1 Application area	Virtual home environment, home networks
2.1.2 Development environment	Rational Rose-RT, JBuilder 4
2.1.3 Platforms	
• Hardware	
• Name	Home device
• Description	Any home device in home environment (containing Java Virtual Machine).
• Software	
• Name	Java Virtual Machine
• Description	The Java Virtual Machine is an abstract computer that runs compiled Java programs.
2.1.4 Interdependencies	Needs Communication Service and Data Distribution Service to form a distribution platform to serve additional upper lever services, e.g. directory and lease services with interfaces to user application.
2.1.5 Prerequisites	
Class	
• Name	public interface Observer
• Description	A class can implement the Observer interface when it wants to be informed of changes in observable objects.
TypeLibrary	
• Name	java.util
• Version	since JDK1.0
2.1.6 Special physical resource needs	N/A

Restrictions

Restrictions should describe all the items that will limit the provider's options for designing or implementing the components. Restriction information is needed when selecting the component for a special purpose. Table 10 defines the restrictions of the example component.

Table 10. The description of the restrictions of the example component.

2.2 Restrictions	Components communicating with LocationService-Manager must extend the class java.util.Observable.

Implementation

Implementation includes composition, context, configuration and interface implementation. Table 11 describes an example of the description of the implementation. Composition information describes the internal structure of the component, which can be derived from the component's class diagram. The component's class diagram must be included, if possible, as well as the classes, operations and attributes. This information is usually available for OCM components only. Context describes the environment of the component, i.e. things that exist or events that may transpire in the environment. Therefore, the context diagram should be included. Configuration defines the variation points where the component behaviour can be changed and the guidelines on how to do it. Interface implementation gives the details of the interfaces. This can be done by including the interface code or referencing the implementation document. In addition, some examples of the use of the interfaces should be included.

Table 11. The implementation information of the example component.

2.3 Implementation	
2.3.1 Composition	LocationServiceManager component consists of six classes: LocationServiceManager, Creator, Observable, LocationOrigin, Observer and BusinessCard. See appendix A: class diagram
2.3.2 Context	See appendix B
2.3.3 Configuration	LocationServiceManager is a core component.
2.3.4 Interface implementation	
• Interface name	N/A
• Implementation (code)	
• Samples of the use	N/A

Delivery

The delivery information provides information about the format of the component when delivering it to a customer. The format can be, e.g., binary format, library format or source code. Delivery time tells the integrator when the component is delivered. The delivery time of the component may be useful to the owner of the product line for tracking versions when adaptation of the component is made by the provider, i.e. in the case of MOTS. Table 12 provides the delivery information of the example component.

Table 12. The delivery information of the example component.

2.4 Delivery	
• Format of delivery	source code
• Delivery time	N/A

4.1.3 Acceptance information

The testing is usually a part of the software verification and validation process (Wallace et al. 1992). Within components, the purpose of the testing information

is to proof the quality of the component. The structure of the acceptance information is presented in Figure 14.

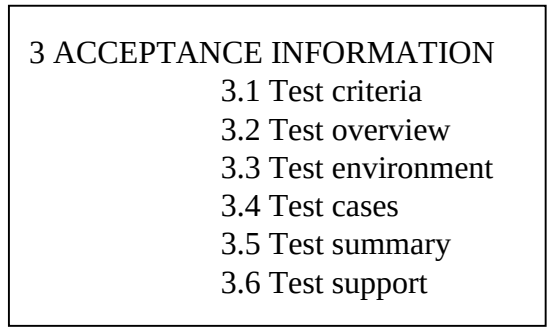


Figure 14. The structure of the acceptance information.

The component must meet the test criteria in order to pass a given test (IEEE 1990). Therefore, the test criteria must be defined beforehand. The test overview gives a short description of the whole testing of the component. The test environment describes the environment in which the component has been tested. The test cases used in testing must be described in order to show how exhaustive the tests have been. For every test case, a short description is given, inputs and outputs of the test are defined and the actual results are compared with the expected results. Anomalies found in the test and the detailed procedure steps are also defined. For follow-up, the testing information should also provide test date and time, the name of the testers and the number of attempts to repeat the test. At the end, a short summary of the test is given. If there are any test support available, they should be defined here for easy evaluation of the component. Table 13 summarises the acceptance information of the example component.

Table 13. The example component's acceptance information.

3.1 Test criteria	For acceptance, the test cases 001, 002, 003 have to be carried out.
3.2 Test overview	LocationServiceManager was tested in the design phase with an executable model. The test cases covered all the situations where a service request could arrive.
3.3 Test environment	Windows NT4.0, Rose RealTime simulation environment
3.4 Test cases	
Test case 001	
• Description	Location service component is asked to generate a location origin object.
• Inputs	void
• Expected outputs	Location origin object containing universal location identifier and physical connections available to contact the location service.
• Actual results	As expected
• Anomalies	None
• Date and time	7.7.2001
• Procedure step	Send signal createLocationOrigin through component interface Creator, no parameters. Receive output as an object.
• Attempts to repeat	One hundred
• Testers	Martti Matinlassi
3.5 Test summary	LocationServiceManager component is able to respond to basic service requests received from a local or a distributed device. This observation bases on the test results of these basic test cases that were carried out.
3.6 Test support	N/A

4.1.4 Support information

The user of the component also needs information on how to install the component and what to do if it doesn't work as expected. Figure 15 provides a structure of the support information. The support information consists of three parts: installation guide, tailoring support and customer support. The necessity of these support services and guides depends on the size and complexity of the component, as well as the form in which it has been delivered. While delivering the component as binary, installation and configuration information are necessary. In the case of source code, tailoring support is needed if the component requires adaptation to the product line.

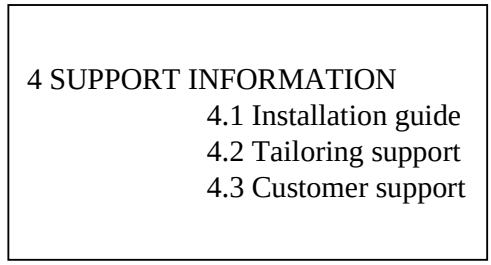


Figure 15. The structure of the support information.

Installation guide

Installation guide defines the operations that must be performed before the component can be used. Table 14 presents a short example of an installation guide. When delivering the component as binary, installation information is extremely important.

Table 14. Installation information of the example component.

4.1 Installation guide	Java classes (source code) of the LocationService-Manager component are inserted into new software system. Generate and compile the whole system.

Tailoring support

Tailoring support defines the persons to contact when help with tailoring the component is needed. Component's tailorability is a property of being adaptable and changeable (Vigder 1998). Especially in the case of source code, tailoring support is needed if the component requires adaptation to the product line.

Customer support

Other support information on the use of the component and possible problems may also be required. Customer support includes the name of the contact person, and an address or a phone number where the customer can get help.

4.2 Development and use of component information

The component documentation model consists of several parts of information. To rationalise the documentation work, the roles responsible for each part of the documentation must be defined. The roles related to component development were described in section 3.3. Each role involved in the component development is responsible for producing a certain set of information pieces for the document. This is how every person involved in the component development documents his own work, and the documents become more reliable and complete.

Figure 16 describes how the roles are related to the component documentation in the case of an OCM component (Niemelä et al. 2002). The component integrator and the component provider co-operate in the documentation procedure. The development and documentation of an OCM component begins on the integrator's side, where the product line architect defines part of the basic and detailed information of the component – the component's requirements specification. A separate requirements specification document is not required as the information can be added directly to the documentation of the component to be developed. The reuse manager validates that the information is in accordance with the product line context. This includes information about the component's purpose in the context of the product line, functional specification, quality requirements and constraints – such as protocols and standards. This information is used by the component architect on the component provider's side, who describes the architecture of the component and complements the basic information.

The component designer provides the detailed information, such as the technical details and technical restrictions of the component, and the design models of the component composition and configuration. The component designer also provides the acceptance test information for the component, such as test criteria, data, methods and cases. The component developer provides the implementation information for the component and complements the acceptance test information by providing the test results and a description of the testing. He also provides the support information for the component. Detailed and acceptance test information is used by the component architect, who validates the component after development and accepts the "component product".

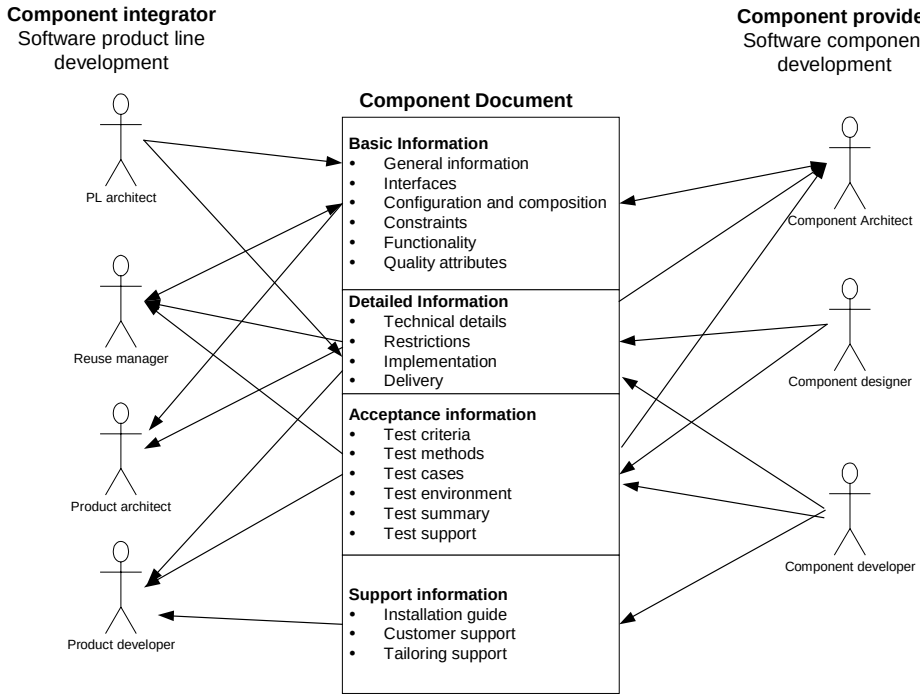


Figure 16. The stakeholders in OCM component development and documentation (Niemelä et al. 2002).

After acceptance, the OCM component is supplied to the component integrator. The reuse manager uses the basic, detailed and acceptance information in the component validation and maintenance. The product architect uses the basic and detailed information when integrating the new component into a new product's architecture. The product developer needs the detailed and acceptance test information when developing a new product.

When developing a COTS component, the component provider is responsible for the whole development and documentation process, as well as in the case of OS components. The component integrator will have prepared a component requirements specification document that he/she now compares with the documentation of candidate COTS components. Figure 17 describes the creation and use of the documentation. The main difference between developing and documenting the OCM component and the COTS component is that the requirements of the COTS component are defined by the markets and the basic and detailed information of the component are provided by a component

architect on the component provider's side. When buying the COTS component, the reuse manager on the integrator's side uses the basic, detailed and acceptance information for the component selection, validation and maintenance. In addition, it is the product line architect's responsibility to perform the analysis of the combined quality attributes of the components and ensure that the component also fits all the other requirements of the product line.

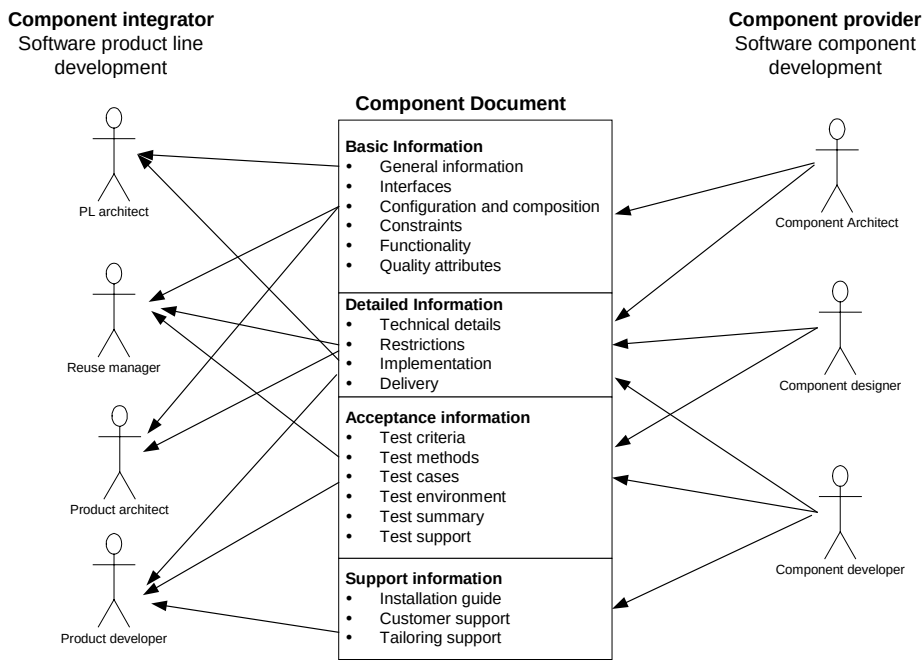


Figure 17. The stakeholders in the COTS/OS development and documentation.

5. Component documentation system

Chapter 4 introduced a component documentation pattern that identifies the information content and structure for component documentation. However, the definition of the pattern is not sufficient for the adoption of a new documentation practice. A new environment that enables the development of the documentation is also required. The environment should provide guidelines concerning what and how to document a software component, and it should also be applicable when, in the case of OCM components, the development of the component is split between two organisations.

This chapter introduces a component documentation system that was built to support the development of component documentation. One of the requirements for the system was that the component documentation does not require much extra work from the component provider but the documentation is easy to develop. In addition, the developed documentation must be consistent with the standard documentation pattern.

The developed documentation system enables the development of component documentation side by side with the component development. The documentation is developed in separate documents following the design phases of the component. Eventually, the separate documents are combined into one document that follows the documentation pattern. The system includes all the necessary tools and technologies for the creation and handling of the documents.

Figure 18 presents the selected tools and developed extensions of the documentation system. Two tools were selected for developing the component documents: Epic Editor for developing textual documents and Rhapsody for developing graphical design models. The component information that the documentation pattern defines is developed in different phases of the component development, and so, for simplicity, the information of the pattern was divided into four separate documents: the basic information is defined in the General Specification document, the detailed and support information is defined in the Detailed Specification and Composition documents, and the acceptance information is defined in the Test Document. The component documentation system provides ready-made document templates, document type descriptions (DTDs) and style sheets for documenting components. It also enables the

inclusion of the design models. In addition, the system provides a tool that combines the four documents into one final component document. The system also enables the viewing of the documents for component buyers, before purchasing (a shorter version) and after purchasing (a larger version).

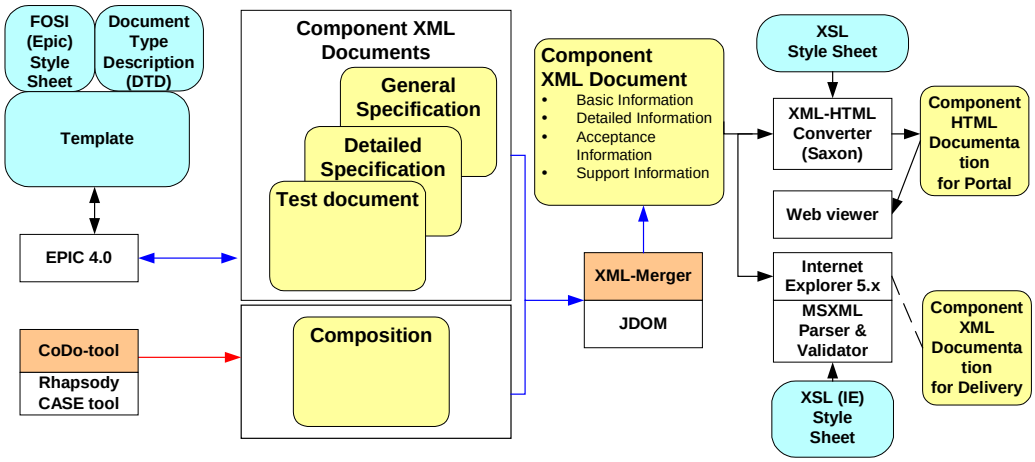


Figure 18. Component documentation system (Taulavuori et al. 2002).

The following sections present the selected technologies, the selected and created tools, and their use in the documentation system in more detail.

5.1 Structured documentation language

Component documents are implemented in the XML format. XML (eXtensible Markup Language) is the World Wide Web Consortium's (W3C) recommendation for a meta-markup language (World Wide Web Consortium 2001a). It was specified as a subset of SGML (Standard Generalized Markup Language) standard, that is an international standard to define the device-independent and system-independent markup notation used in structured documentation. Due to SGML's complex technical documentation, it has been found to be too difficult for serving documents over the web. XML was created to combine the flexibility and power of SGML with the widespread acceptance of HTML (HyperText Markup Language), having good features of both of them (Walsh 1998). HTML is an application of SGML used in creating and displaying web pages.

The main difference between HTML and XML is that HTML does not provide a mechanism for describing the content of the document. HTML is said to be human readable because it only offers information on how the document is displayed with the fixed markup tags. XML is computer readable because it provides information on how to describe the document content, structure and meaning. Therefore, it enables platform-independent data exchange between applications (Walsh 1998; World Wide Web Consortium 2001a). The extensibility and self-describing nature of XML means that users can define their own set of markup tags. These tags must be organised according to certain general principles of a Document Type Description (DTD), which specifies the rules for the structure of a document. Unlike HTML, XML does not include any formatting instructions. Formatting can be added into XML documents with a style sheet (Harold 1999).

XML was chosen as the format of component documentation because of its advantageous qualities. XML allows easy definition of the document content and rules for the structure of documents. An XML document consists of semantic tags (elements) that break a document into parts and identify the different parts of the document. Because of XML's self-describing nature, it is easy to define the document elements so that the element names reflect the meaning of the elements. For component documentation, the elements can be defined to describe the characteristics of the software components. For example, the element "Interface" can be created and used to describe the interface of a component. The order and appearance of the elements can be defined by the rules declared in the DTD. It can be specified, for example, that every "Interface" element has a name-attribute and a child element called "Description". The rules ensure that the documents are consistent.

XML was chosen in component documentation because it also enables document interchange between different applications over the web. In addition, XML is a non-proprietary format and is not encumbered by any sort of intellectual property restriction. Any tool that understands XML format can be used to handle XML documents. The most common tools in an XML document life cycle are XML editors, parsers and browsers (Harold 1999). The XML document is usually written with XML editor. The XML parser is a tool that reads the document and converts it into a tree of elements. The parser also checks that the document is valid and well-formed. Validity means that the

document matches the constraints listed in the DTD. Well-formed document denotes that the document adheres to the rules set for the XML language (Walsh 1998). The parser passes the document tree to the browser that displays the document to the user. To be able to display the document, the browser needs the style sheet which tells the browser how to format individual elements.

5.2 Textual documents

Epic Editor 4.0 is an XML editor that enables easy creating and processing of XML and SGML documents. Epic Editor can be executed with several versions of Windows NT and Sun Solaris (Arbortext 2002).

Epic Editor is used here to develop General Specification, Detailed Specification and Test Documents. The content and structure of component documents are defined beforehand, so that the developed documents are consistent and contain the required information. The definition of the content determines what information must be included in the documents. The definition of the structure determines the rules that the documents must adhere to, i.e. how the information in the documents is structured and how the pieces of information relate to each other. To enable consistency of the component documents, and to ease the document writer's work, the DTDs, document templates and style sheets have been developed for each of the component documents.

DTD (Document Type Description) specifies the elements, attributes and entities contained in the XML document and their relationships (Harold 1999). The document rules of the DTD specify, for example, which elements and attributes are compulsory in a document and which are optional, how many times a certain element can appear in a document, and the order of the elements. The following shows a simplified example of the definition of component identification elements in the DTD:

```
<!ELEMENT Component_id - - (Identification, Name)>
<!ELEMENT Identification - - (Text)>
<!ELEMENT Name - - (Text)>
<!ELEMENT Text - - (#PCDATA)>
```

DTDs for component documents are defined especially for Epic Editor. Epic Editor's DTDs have small differences in comparison with the DTDs developed for Internet Explorer, for example.

Document template is a base form for the document to be written. The template contains elements from the DTD that are normally required in any instance using that DTD. It shows the writer all the needed elements and their place in the document. The documentation system's templates also include writing instructions. These instructions tell the writer the meaning of the element, i.e. what information should be provided for each element. The template ensures that all the elements are in their own places in the document and no obligatory elements or attributes are missing. The following shows a piece of document template in the XML format:

```
<component_id>
<identification>
<text>[A unique identification number of the component]
</text>
</identification>
<name>
<text>[A unique name of the component]</text>
</name>
</component_id>
```

The writing instructions for the user are defined between square brackets. Epic Editor enables the development of documents in a graphical editing view, so the user does not see the documents in tag format.

Epic Editor supports the FOSI (Formatting Output Specification Instance) style sheets that specify the outlook of the documents (Arbortext 2000). A style sheet contains the modification instructions for all the elements and attributes defined in the DTD. Because the users cannot affect the modification instructions of the style sheets, the outlook of the documents is always consistent. Furthermore, the headlines for the document paragraphs are defined in the style sheet so that the user will not be able to change the structure of the document.

Figure 19 presents the document editing view of General Specification in Epic Editor. When creating a new XML document with Epic Editor, the user receives a template document based on a certain DTD. As the user has the attributes and

elements defined by the DTD, he can write the document following the guidelines. He can also add elements into a document according to the DTD. Epic Editor's completeness check ensures that the rules for the document structure defined in the DTD are being adhered to (Arbortext 2000). For example, the user cannot remove elements that are being defined as compulsory in the DTD. Any completeness error is declared to the user.

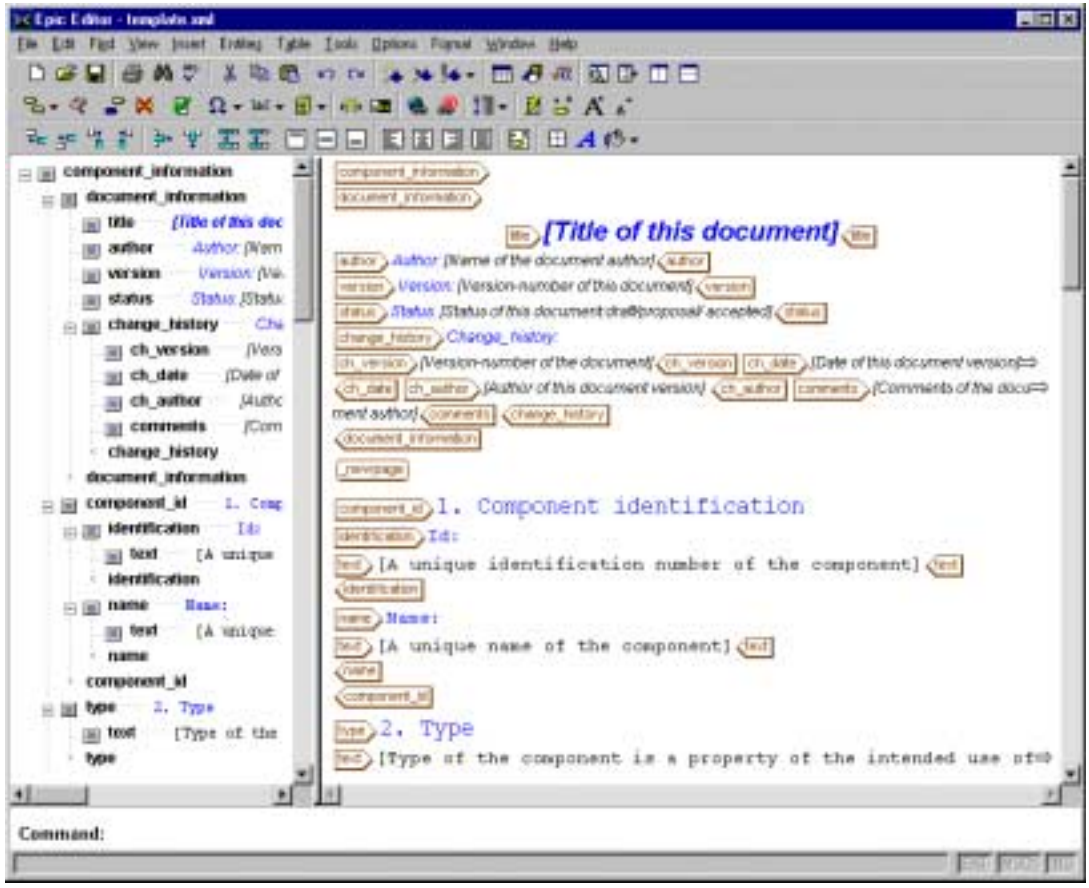


Figure 19. The document editing view of Epic Editor.

5.3 Graphical documents

The CoDo tool is a documentation tool that was developed to convert data from a component's class diagram into the XML format. The CoDo tool is an

extension software component to Rhapsody 2.2 or greater. Rhapsody is a Unified Modelling Language (UML)-compliant visual design tool for developers of real-time embedded software (iLogix 2002). UML is a widely applied description language in architecture and component design. Rhapsody API allows users to programmatically interact with Rhapsody projects for useful applications, such as the preparation of custom reports.

The Rhapsody API functions through a set of methods and attributes that act as a set of Microsoft COM (Component Object Model) interfaces. COM is MicroSoft's software architecture that allows the components developed by different software vendors to be combined into a variety of applications (Microsoft 2002a). COM defines a MicroSoft standard for component interoperability and it is not dependent on any particular programming language. It is available on multiple platforms and it is extensible.

The CoDo tool is implemented by the Visual Basic programming language and is attached to Rhapsody with help of the COM interface. Figure 20 shows how the XML file is developed from the Rhapsody class diagram. The CoDo tool reads data from a Rhapsody file with the help of the Rhapsody API and inserts information into a DOM (Document Object Model) tree. DOM is a specification for platform- and language-neutral application program interfaces for accessing the content of HTML and XML documents (World Wide Web Consortium

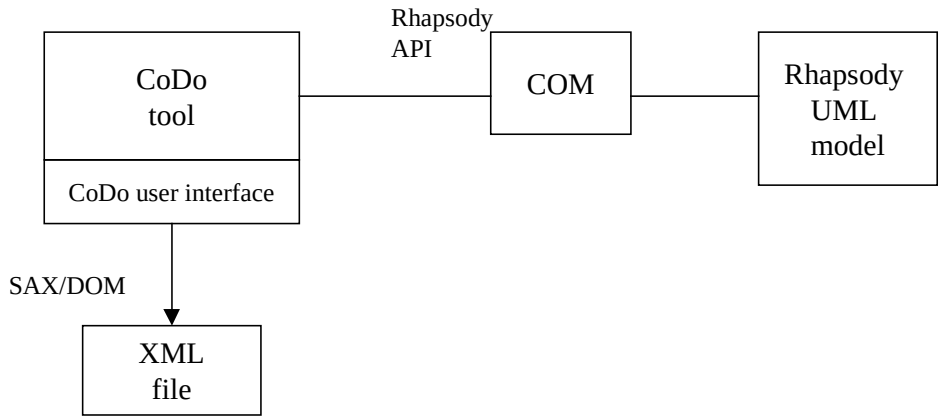


Figure 20. CoDo tool's logic.

2002a). The DOM tree is saved into the output XML file using the MSXML parser. MSXML is Microsoft's XML parser that is accessible from many programming and scripting languages, and can be used in any application to access XML files (Microsoft 2002b).

The CoDo tool develops an XML document about the component's composition. Figure 21 shows CoDo tool's user interface. The user can include document information (title, author, version and status), component identification number and component name into the document with the user interface. The name of the Rhapsody project file is required in order to obtain the class diagram information from Rhapsody, e.g. classes, attributes and operations, and their descriptions.

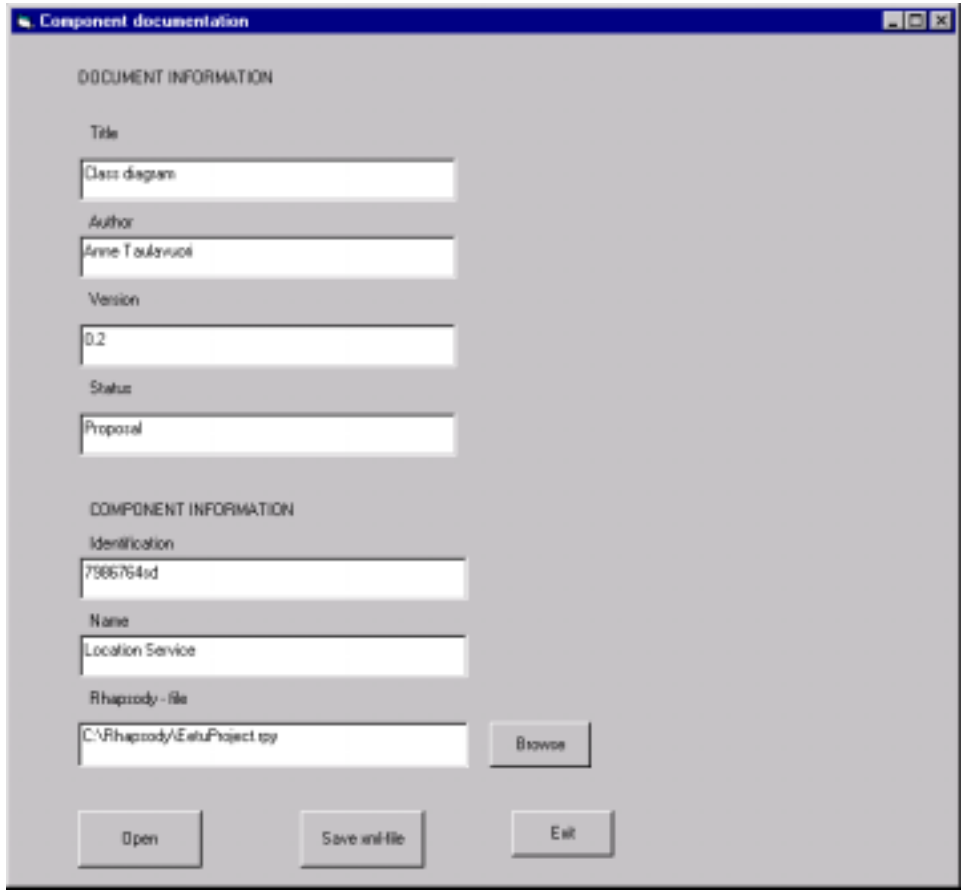


Figure 21. CoDo tool's user interface.

5.4 Integrated documents

The XML merger is a tool that was developed to combine the separately developed component documents into one document (Taulavuori et al. 2002). The XML merger is implemented by the Java programming language and it uses Java Document Object Model (JDOM) for handling XML data. JDOM is an open source Application Programming Interface for accessing the content of HTML and XML documents (JDOM Homepage 2002). It is a Java representation of an XML document and provides a way of representing a document for easy and efficient reading, manipulation and writing (Harold 2001). It is an alternative to such existing standards as Simple API for XML (SAX) and the Document Object Model (DOM), although it integrates well with both of them.

JDOM models an XML document into a series of Java objects (Biggs & Evans 2001). These objects are then used like any other Java object to manipulate and modify an XML document. All XML building blocks – such as elements, attributes and processing instructions – have a corresponding class in JDOM (Harold 2001). Using an underlying SAX or DOM parser, an XML document is parsed and represented as a Document object (Biggs & Evans 2001). JDOM presents an XML document in a tree form, where the root element is the key to accessing the rest of the document. Since Java naturally supports object inheritance, it is possible to navigate the object hierarchy representing an XML document and extract the wanted information.

The XML merger combines the four component documents (General Specification, Detailed Specification, Composition and Test document) into one XML document and thus builds up the final documentation of the software component. The XML merger gets the file names of the four documents as inputs, combines the documents and modifies them, and creates one output file.

The XML merger creates a new root element, Master_document, for the combined document, and adds the four documents with their child elements. Child elements are elements that are included in another element (i.e. parent element). Figure 22 describes the main structure of the combined document.

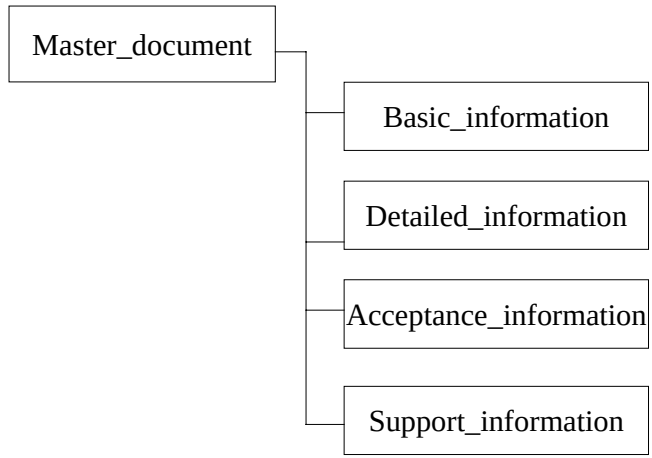


Figure 22. The combined component document.

XML merger modifies the document so that the final documentation follows the document pattern defined in chapter 4. Some modifications have to be made because of the incompatibility of different tools. In addition, some processing instructions have to be added in order to enable the user-friendly viewing of the XML document.

The XML merger executes the following steps while merging documents:

- Eliminate Epic Editor specific processing instructions.
- Create the structure of the document that is consistent with the standard documentation pattern.
- Remove the duplicate information.
- Attach headlines.
- Insert new processing instructions in the merged document.

When all the modifications are finished, the XML file is created and the JDOM tree is saved using JDOM's XMLOutputter. XMLOutputter takes a JDOM tree and formats it to a stream as XML (JDOM Homepage 2002).

The JDOM beta 6 source is free and the licence enables developers to use JDOM in creating new products without requiring them to release their own products as

open source (JDOM Homepage 2002). The XML merger was developed using JBuilder 4.0.

5.5 Viewing of documents

The documentation system takes into account the possible trading of the component, and that is why the system has two different versions of the final component document: a component document for portal and a component document for delivery. The document for delivery is the final component document that is sent to the customer when buying the component. The XML document can be viewed with a web browser using the XSL style sheet. XSL (Extensible Style Language) is a style language for presenting structured content – i.e. styling, laying out and paginating the source content onto some presentation medium, such as a web browser (Harold 1999). XSL is divided into three sections: transformations, formatting and expression (World Wide Web Consortium 2002b). The transformations section enables replacing tags, adding additional content into XML documents and reordering the elements of the XML document. The formatting section enables specifying the appearance and layout of documents as pages for web browsers. The expression section, XML Path Language (XPath), is an expression language whose primary purpose is to address parts of an XML document.

To be able to view an XML document with a web browser, the XML document has to have a reference to an existing XSL style sheet. An XSL style sheet contains templates into which data from the XML document is poured. The web browser tries to match parts of the XML document with each template element of the style sheet, and when the match is found the XML document part is processed by the rules of the style sheet (Harold 1999).

The document for the portal is a cut version of the final document. It is placed on the portal's www page and is converted to the HTML format. The XML document is transformed into HTML using XSL Transformation (XSLT). XSLT is a part of XSL and is used for transforming XML documents into other formats or XML documents (World Wide Web Consortium 2002b). The XSLT processor is the main component of the SAXON package, which is a collection of tools for processing XML documents (Kay 2001). The XSLT processor

implements the Version 1.0 XSLT Recommendations of W3C. The portal document only shows customers the component's basic information and technical details for browsing the components. The rest of the document – mainly the information concerning the design and implementation details – is hidden from the customer with the help of the XSL style sheet.

6. Applying the component documentation pattern

The developed component documentation pattern was applied in practice in three ways. First, the documentation pattern was validated by an experiment that simulated a situation where the component provider develops documentation for a software component following the defined documentation pattern. This enables the viewing of the documentation pattern from the perspective of the component provider while developing a component for-reuse. Next, the pattern was validated by industrial software engineers. The engineers viewed the pattern from the viewpoint of the component integrator in a with-reuse process. After that, the developed component documentation pattern was compared with the documentation of selected COTS and OS components.

6.1 Pattern analysis from the viewpoint of a component provider

The documentation pattern was applied by developing the documentation system and documenting an example component. The required documentation tools were implemented and the whole documentation process of an example component was simulated.

6.1.1 Development of the documentation pattern and system

Overall, the development of the pattern – i.e. the content of the documentation – turned out to be the most difficult phase. The literature about the components and the product lines was carefully examined to discover the important concepts of components from the viewpoint of product lines. The discovered documentation requirements were classified and arranged in a logical structure. Still, the content and the structure were modified many times afterwards and several new documentation requirements arose during the system development. Thus the pattern evolved right up to the last moment.

XML was chosen for the document implementation technology because it provided several possibilities. The development of the CoDo tool was seen as necessary because the graphical documents were to be included in the component documentation.

The Rhapsody CASE tool was chosen as the tool for the graphical documents because it was already familiar. There were two different ways of expanding Rhapsody. These alternatives were the use of Rhapsody's COM interface or the use of Rhapsody's save file. In the latter alternative, the UML models would have been read from the Rhapsody's save file, which is a text file, and then converted to an XML document. However, Rhapsody's own save format was not publicly specified, so the additional software component could not be directly integrated with it. The use of the COM interface was chosen because it was the more effective and simpler alternative. The COM interface at Rhapsody is designed especially for building these kinds of additional software components and it includes methods for support reading information about UML models.

When choosing the tool for creating textual documents, two XML editors were compared: XMLSpy 3.5 and Epic Editor 4.0. Because the technical comparison between these tools did not indicate the superiority of XMLSpy, Epic Editor was selected as it was already available. The requirements for the documentation were that the development of the component documentation should be as easy and straightforward as possible for the users, and that the users could develop the documentation fluently together with the component. In the end, it was decided to create four different document templates that would follow the design phases of the component development. This kind of document definition is also practical when documenting OCM components between the integrator and the provider.

The development of the tool that combines the documents was necessary because, for the sake of simplicity, the component documentation was defined to be one concrete document that would cover all the information about the component. The different ways of combining the four XML documents of the component were considered. First, the combination was tried using XML technology and the entity references of the DTDs. It came out that XML and web browsers were not mature enough for this kind of use. JDOM was chosen as the integration technique due to its advantageous qualities. Using JDOM, the

combination could be done by using Java code, and JDOM also has clear instructions on how to modify and remove elements and attributes of XML documents. The XML merger was built with the help of JDOM, which offered easy access to the content of the XML document.

When the structure of the combined XML document was specified, the XSL style sheet for the viewing the document with web browsers could be developed. The style sheet was combined with the XML document with the help of the XML merger, which adds the style sheet reference to the XML document. After that, the documentation is ready and can be uploaded to the portal with the component.

6.1.2 Testing of the documentation system

The development of the component documentation begins at the same time as the software architecture design. The component provider has to have the following tools: Epic Editor for developing the textual documents, Rhapsody with the CoDo tool for developing the graphical documents and transforming them into textual format, the XML merger for combining the documents, and the XSL style sheet for viewing the final documentation.

The documentation system was tested by developing a document for an example component using the developed documentation system. Figure 23 illustrates the use of the documentation system. The development of a component was started by defining the requirements for the component. The document template of the General Specification was fulfilled with Epic Editor by the component architect. The architect also fulfilled the technical details in the Detailed Specification document. When defining the requirements, the architect started the planning of the component test by defining the component test criteria for the Acceptance Information document.

In the design phase, the UML diagrams of the component were drawn with the Rhapsody CASE tool by the component designer. When the design diagrams were ready and approved, the designer developed the Composition document with the CoDo tool. The other detailed information about the component design and implementation was complemented by fulfilling the Detailed Specification

document with Epic Editor by the component designer and component developer.

When the component was implemented, it was tested normally. The component designer and the component developer fulfilled the Acceptance Information document when creating the test cases and when doing the testing. When the implementation was being approved, the Support Information was complemented by the component developer.

When all four component documents were ready and approved, the component architect approved the component product. The architect combined the documents into one document with the XML merger. The file names of the four documents were given to the XML merger as inputs. After executing the XML merger, the documentation of the component was ready and could be viewed with the browser, as the XSL style sheet was being attached.

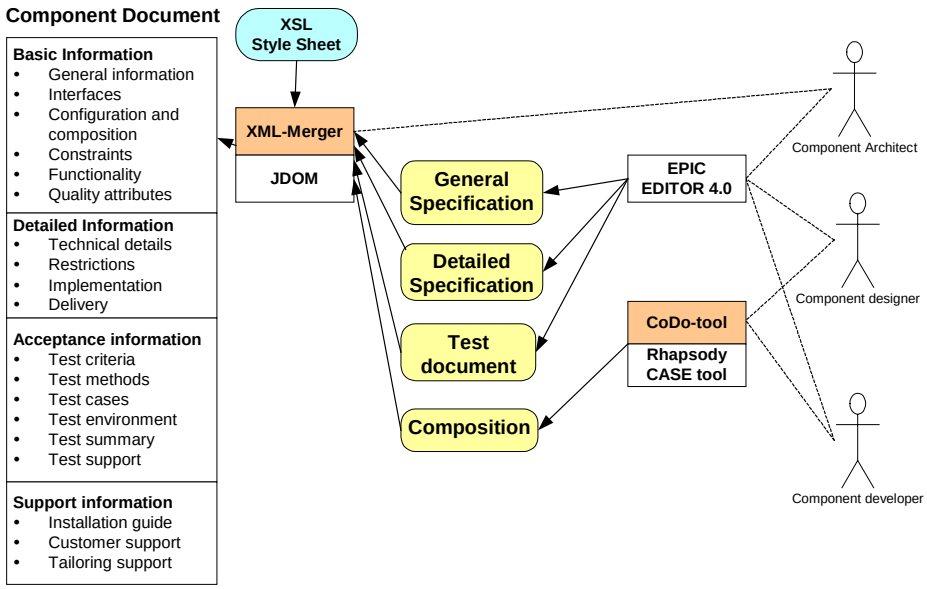


Figure 23. The use of the documentation system.

In summary, the documentation of an example component went fluently. The document templates and the CoDo tool guided what information was to be provided for the component. The user only had to follow the instructions. Epic

Editor's document editing view was clear and the completeness check function ensured that the user could not make any completeness errors in the document. The XML merger ensured that the structure of the combined document was the same as the structure of the documentation pattern. The user did not have to know what the final structure would be. The user only fulfilled the template documents and the system did the rest.

6.1.3 Experiences and further work

At the time the documentation system was developed, all web browsers did not support the XML technology. These XML documents were implemented for use with Internet Explorer 5.5. The IE browser must include at least release 3.0 of the Microsoft® XML Parser (MSXML). Nowadays, Netscape 6.x also supports XML.

The developed documentation system provides some major benefits. The documentation environment provides a systematic method and guidelines for documenting the properties of software components. Component documentation will be consistent when following the guidelines. The analysis showed that it is possible to produce data for documentation easily while designing the details of component. Documentation is easy to prepare with Epic Editor and the ready-made DTDs, templates and style sheets. The tool also provides the user with a clear view of the document and checks the completeness of the XML documents. The Rhapsody API enables the inclusion of UML diagrams into XML documents. In addition, the documents can be manipulated with ease with the help of JDOM, which is efficient in handling XML documents. JDOM is designed especially for Java programmers and it is easy to learn and use. However, in order to improve the documentation system, the document management system is required for the variants and versions of component documents in the maintenance of a product line.

There were also shortcomings that came up during the development of the component documentation system. These shortcomings were mostly caused by the immaturity and incompatibility of the tools. The XML technology turned out to be still immature, as the system required several modifications. The DTD and style sheet definitions of the IE and Epic Editor were not compatible. This

implies that documents developed with Epic Editor cannot be viewed directly with IE. Three style sheets of different types were needed, one for the Epic Editor (FOSI), the other for viewing documents with Internet Explorer and the third for converting the XML documents into HTML format for the web site. The diversity of the tools required a lot of work and tailoring that should not be necessary when using an efficient integration language. However, as the XML technology is maturing rapidly, these technical problems may soon be resolved.

The use of heterogeneous CASE tools brought about several difficulties and additional work due to a lack of ability to interchange design information between the CASE tools. In the autumn of 2000, when the development of the documentation system was started, there was no support for XMI (XML Metadata Interchange) in any CASE tool, which created the need to develop the CoDo tool. The XMI standard allows data exchange between tools, applications, and repositories over the web (Object Management Group 2002). The XMI standard combines the benefits of the web-based XML standard for defining, validating and sharing document formats developed with object-oriented Unified Modelling Language (UML). The newer version of Rhapsody, Rhapsody 3.0, includes an XMI interface as a standard component, allowing developers to exchange designs from other UML tools (iLogix 2002). The XMI interface enables developers using Rhapsody to both export UML meta model information from Rhapsody and import UML information from other products which support the standard. Used in the documentation system, the XMI standard enables the transfer of all component design data from providers to integrators. But, as more design data is available, this can mean refining the documentation pattern.

6.2 Pattern analysis from the viewpoint of a component integrator

The interviews regarding component documentation were conducted in two software companies during May 2002. The main purposes of the interviews were to ascertain the most important information to be documented for a component, and obtain an expert assessment of the developed documentation pattern. The most important requirement when selecting the interviewees was that they must

work with software components, either developing components or building products by using components.

The questions asked in the interviews concentrated on the requirements of component documentation: What information should the component documentation include and why is this information important. Appendix A details the questions.

One of the aims of the interviews was discover the problems that the software engineers had found when dealing with the components, especially when selecting the appropriate components and when integrating the components into their systems. The engineers were also asked how these problems could be solved with the help of documentation, and what the state of component documentation is today. The answers were based mostly on the engineers' experience; when they could not directly answer the question, they were asked to give estimations.

6.2.1 Requirements for the documentation

All the engineers interviewed agreed that one of the major problems when dealing with components is insufficient documentation. This makes the validation of the component's quality particularly difficult. The proper documentation was seen as the key to successful component selection and integration. The component documentation was required to be clear, extensive and up-to-date. In addition, all documents should be consistent – i.e. "universal" – which would make information retrieval easier. The quality of the documentation is directly related to the quality of the component itself. There should not be any features of the component that are not documented and all the features should be documented with the same accuracy. The documentation should be specific enough, so that the execution environment of the component could be implemented beforehand. One of the most important requirements for the documentation was that it must be completely consistent with the component itself.

The interviews revealed that the component documentation issues are not mature enough in many software companies today. Documentation is often product-

specific, not yet component-specific. In addition, there is no universal documentation practice in use, but the documentation is guided by the organisation's own quality manual. The problems with components can be solved inside the organisation informally, simply by asking. When using commercial components, the problem solving is not so easy. The documentation of components is now one of the key issues in many software companies, especially when acquiring components from outside the organisation.

6.2.2 Most important issues

According to the interviews, one cannot ever be sure if the component is applicable to a certain product architecture, even if the interface descriptions are available. The applicability of the component can only be verified with the help of component prototype or testing. Therefore, it is extremely important that the execution environment of the component is documented. The execution environment covers all the critical conditions of the component, such as platforms, interdependencies, physical resource needs and other possible prerequisites.

All the engineers stated that the most important information regarding a component is the definition of the component interfaces. The interface information is critical, both in selecting and integrating components. For selection, the interface definition may be adequate for the user to determine whether or not the component is applicable to his purpose and environment. For integration, however, the interface implementation is fundamental for a user to be able to connect the component with other components of the system. The interface implementation must be provided at the code level, and the examples of the use of the interfaces are necessary in order to use them correctly. The configuration issues and variation points should also be defined in connection with the interfaces.

Functionality also seemed to have a great emphasis among the interviewees, because it describes exactly what the component does. UML, especially use-case and sequence diagrams, were seen to be the best possible way of describing the functionality of the component. In addition, it became clear that it is sometimes

necessary to provide a short, verbal overview of the functionality to give a brief view of the component.

The quality of the components had an essential position in the interviewees' organisations. In most cases, the quality is defined with the help of common quality attributes. The most important quality attributes include modifiability, including maintainability, and performance. Expandability and reliability also had great status according to some of the engineers. Quality requirements often have their own part in the requirements specification.

In order to be able to validate the quality of a component, the interviews showed that the component documentation should include the component's test report. The test report is often the only way of knowing how the quality of the component is verified. The test report should include the test cases for all the quality and functional requirements of the component, and also cross-references between the requirements and the test cases. In addition, all the critical exceptions of the component should be noticed in the test cases. The test report should be so specific that it is possible to repeat the test according to it. As noted, it is often impossible to build all the test cases for COTS components because their black-box nature.

The interviews revealed some differences between COTS and OCM component documentation. The best documentation for an OCM component is the source code. In addition, there should be more specification and design documents for an OCM component so that the users are able to modify the component and exploit it in the future. According to the engineers, modification of the OCM component can be implemented with the help of the information on interfaces or architectural design. OCM documentation includes information about the integrator's own cases and environments, because the OCM integrator affects the component when defining the requirements for the component. Thus the architectural requirements of the integrator, such as protocols and restrictions, are critical for OCM components, as some of the engineers stated. The documentation of a COTS component must be extremely specific, because the component is used without access to the source code and, therefore, any problem solving is more difficult.

In summary, it can be stated that the developed documentation model responds pretty well the software engineers' insights about the documentation requirements of the component. The most important information about components that all the engineers were agreed on was interfaces, functionality, modifiability, performance, testing and execution environment. Also, expandability, reliability, restrictions, protocols and configuration were seen as quite important. In addition, the architectural design was seen as fundamental for OCM components.

6.2.3 Improvements

For component selection, the user needs all the possible information available. The engineers agreed that the user should be able to view the whole documentation of a COTS component before purchasing it. There were also some new, suggested documentation requirements that came out in the interviews. If the component already has some integration components, they should be mentioned in the component documentation. The life cycle of the component can give useful information in helping to estimate the component quality. It may define, for example, the maturity of the component and any possible variants. Some kind of estimation about the degree of the reusability of the component would also be useful when searching components.

At the end of the interviews the software engineers were shown the developed component documentation pattern described in chapter 4. The engineers were asked to assess the pattern and give improvement suggestions. Some of the suggestions concentrated on the documentation of the example component, not on the documentation pattern. The following summarises the most significant replies and suggestions:

- Examples of the use of the interfaces could be added.
- The description of the functionality could be provided with use cases.
- The component execution environment could be mentioned earlier in a document, because it is essential when considering the selection of the component.

- The documentation should have the information fields for the possible integration components and the life cycle of the component.
- The critical exceptions and the quality requirements of the component should be noticed in the test cases.
- The information in the class diagram is very useful. The arguments about the functions should be described more specifically and their meaning should be made clearer.
- The structure of the document is clear.
- The document is comprehensive.
- If a component is documented following this pattern, the documentation is adequate for component selection and integration.

6.3 Pattern comparison with documentation practices

The developed documentation pattern was also compared to the documentation of example components picked at random from the Internet. The purpose was to find out how current component providers have actually documented their components and how these documents relate to the documentation pattern.

Four example components were searched and their documentation was evaluated on the basis of the documentation pattern. The first component, MySQL, is a database that represents OS components and is freely downloadable from the address "http://www.mysql.com/". The second component, MicroSoft Research IPv6, is a special case of OS components with a specific, restricting licence. The MSR IPv6 is downloadable from the address "http://research.microsoft.com/msripv6". The third component, QuickXML, is an example of a COTS component and can be bought from the component marketplace "http://www.componentsource.com". The fourth component, Java Image Converters, is also a COTS component and can be bought at "http://www.flashline.com". The results of the documentation evaluation can be found in Table 15.

Table 15. Evaluation of the documentation of the example components.

Required information	MySQL	MSR IPv6	Quick XML	Java Image Converters
1 Basic information				
1.1 General information:				
· Component id number	-	-	-	-
· Name of the component	x	x	x	x
· Type of the component	x	x	x	x
· Overview	x	x	x	x
· History				
· Version	x	x	-	x
· Developer	x	x	-	-
· Time of the actions	-	-	-	-
· Improvements	x	x	-	-
· Special terms and rules	-	-	-	-
1.2 Interfaces:				
· Version usage	-	-	-	-
· Provided interfaces	x	x	-	-
· Required interfaces	-	-	-	-
1.3 Configuration and composition	x	x	-	
1.4 Constraints:				
· Protocols	x	-	-	-
· Standards	x	x	-	-
1.5 Functional specification:				
· Name	x	x	x	-
· Description	x	x	x	-
· Inputs	x	-	-	-
· Outputs of the component	x	-	-	-
· Functional exceptions	-	-	-	-
1.6 Quality attributes:				
· Modifiability	-	-	-	-
· Expandability	x	x	-	-
· Performance	x	-	-	-
· Security	x	x	-	-
· Reliability	x	-	-	-

Continues

Table 15 continued. Evaluation of the documentation of the example components.

2 Detailed information				
2.1 Technical details:				
· Application area	-	-	-	-
· Development environment	x	x	x	x
· Platforms	x	x	x	x
· Interdependencies	-	x	x	-
· Prerequisites	x	x	-	-
· Special physical resource needs	-	-	x	x
2.2 Restrictions	x	x	-	x
2.3 Implementation:				
· Composition	-	-	-	-
· Context	-	-	-	-
· Configuration	x	x	-	-
· Interface implementation	-	-	-	-
2.4 Delivery	x	x	x	x
3 Acceptance information				
3.1 Test criteria	-	-	-	-
3.2 Test overview	-	-	-	-
3.3 Test environment	-	-	-	x
3.4 Test cases	-	-	-	-
3.5 Test summary	-	-	-	-
3.6 Test support	x	x	-	-
4 Support information				
4.1 Installation guide	x	x	x	x
4.2 Tailoring support	x	-	-	-
4.3 Customer support	x	-	-	x

As it can be seen from Table 15, of all the four examples, the first open source component is the best documented. Generally, the documentation for open source components is more widely available. The technical details and restrictions are the best documented of the four components. The description of functionality is provided in the form of function overviews; the specific descriptions are missing in three cases. Also, the general information is quite freely available, although the design rationale and design rules are missing. The quality and testing were the weakest documented. The quality of the component was dealt with only in two cases; both were OS components. To be able to make a buying decision, the user needs all the possible information about the

component. The documentation of the sample components, especially the COTS components, was inadequate for this purpose. The integrator can compare the technical details but he cannot be sure if the component is applicable to his product line.

When examining the documentation of components generally, two observations could be made. Firstly, components, especially COTS components, are insufficiently documented and the quality of the documentation varies considerably. Secondly, even if the required information is provided in component documentation, this information is difficult to find. The structure of much of the documentation was confused and it took too long to find the required information. Another observation was that there was too much documentation for some components, and in those cases locating the most essential information was difficult. Thus, a consistent structure for the documentation can be seen as fundamental when comparing components and searching information from the documents.

6.4 Summary

The analysis of the documentation pattern and system from the viewpoint of the component provider showed that the pattern and system has some major benefits. The provider is able to create documentation while developing the component and, with the ready-made tools and document templates, he has clear guidelines on how to document components. However, the used technology and tools were not yet mature enough. The new standard, XMI, enables several improvements to the documentation system, and thus it has influences that may require refinement of the pattern as well.

In order to use the documentation system, the provider has to acquire the required tools. In the same way, the component integrator has to have the required tools for documentation when ordering OCM components. At this time, the documentation system includes several tools that the document provider needs.

Analysis of the pattern on the basis of the interviews showed that the pattern is comprehensive and clear for component documentation. Still, the interviewed

software engineers suggested some improvements on the basis of their experience of using components. The suggestions did not, however, change the basics of the pattern but concentrated mainly on information presentation and the adding and specifying of some information fields.

Both the interviews and the analysis of the sample documentation revealed that the current documentation of components is still inadequate. The component integrator cannot get specific information about the internals and quality of a component. When examining the sample documents for COTS and OS components, it is instantly noticeable that the documentation quality varies considerably from one component provider to another. A standard documentation pattern is necessary to enable the consistency and sufficiency of the component documentation.

7. Conclusions

The lack of a standard component documentation model has been seen as a bottleneck in both today's and the future's component trading. Proper documentation is the only way to verify the capabilities of a third-party component, and assess its applicability to an architecture. The difficulties in assessing components is emphasised within software product lines, in which the common architecture affects on several component-related decisions. The product line architecture exposes the critical infrastructure elements of the component and imposes the role of the component.

The purpose of this study was to define the documentation requirements of software components from the viewpoint of product lines and create a documentation model for components from these requirements. Therefore, all the capabilities of a component that are required to be known when assessing the applicability of a component for a product line had to be identified. When examining the use of components in product lines, several documentation requirements for components could be identified. These included, for instance, information about the component's quality and constraints – such as protocols and standards, communication, functionality, variability, and other architecture-specific rules. The defined documentation requirements of a software component are presented in this study in the form of a standard documentation pattern – that is, a skeleton for component documentation.

The developed documentation pattern defines the information content and the structure of the documentation. The pattern is divided into basic, detailed, acceptance and support information. The basic information describes the component's general properties and responsibilities from the architectural point of view, and thus relates to the component requirement specification. The detailed information provides a detailed description of the component's design and implementation. The acceptance information corresponds to the component acceptance test, proofing the quality of the component. The support information part provides information on the use and maintenance of the component.

The documentation pattern has advantages for both component providers and integrators. The pattern provides clear guidelines on how to document the properties of a component, and thus guarantees the consistency and quality of

component documents. By identifying the roles in the component development, it can be defined who is responsible for each piece of information in the documentation. When defining the roles, both on the component provider's and the integrator's side, and associating them to the pattern, the development of an OCM component documentation can be described.

Documentation that follows the pattern assists component integrators in searching, selecting, validating, integrating, using and maintaining the component. The documentation structure and implementation technique supports information retrieval, thus assisting the component search. For the selection of the component, the integrator needs all the possible information available. Following the pattern, the documentation provides information on the component’s functionality, interfaces, quality, environments, architectural rules and constraints, which assist integrators to decide if the component is applicable to their purposes. The documentation also helps integrators to validate the component, providing information about how the component fulfils its requirements and how the quality of the component has been verified. Detailed information about the component’s interfaces helps integrators integrate the component into their systems. The user's guide and information on the component's restrictions, required resources and configuration enable the use of the component. In addition, design models and the information about the component's implementation assist integrators in component modification and extension. Customer and tailoring support are required in component maintenance.

The definition of the pattern is not merely sufficient for the adoption of a new documentation practice, which is why the system to support the development of the documentation was developed. The system provides the technology and tools that enable the development, handling and viewing of the component documents. It also ensures that the documentation is in accordance with the defined documentation pattern.

In the documentation system, the documents are implemented with XML technology, which provides the platform-independent data exchange between applications and the easy definition of the document content and the structure. The developed document templates, DTDs and style sheets enable the easy creation of component documents with an XML editor. A developed extension

tool integrated with a commercial CASE tool enables the transformation of UML diagrams to the XML format. The developed merger tool combines the separately created component documents, modifies them and thus creates the document structure that corresponds to the structure of the developed pattern. The style sheets enable the modified viewing of the XML documents with the web browsers.

The documentation pattern was applied in an experiment using the developed documentation system that produces the intended documentation. Experiences showed that it is possible to produce documentation data easily while developing the component. The system’s document templates follow the design phases of the component, when the development of the documentation is easy and does not require extra work. The development of the documentation system also declared that today's technology is mature enough for building such a system, although some incompatibilities were found between the tools. The interviews with the software engineers revealed that a standard documentation pattern is required. The interviews summarised that the developed pattern is comprehensive, clear and adequate, although some minor modifications could still be made. Comparing the pattern with the sample components' documentation revealed that the pattern provides more information on several aspects of the components than recent component documentation models.

The documentation pattern provides a standard form for describing components, which was seen as a prerequisite in component trading. When using the pattern, a bottleneck in the component development, acquirement and utilization is removed. The developed documentation system enables the use of the pattern. To achieve total benefit from the pattern, the commitment of the various component providers and component integrators is still required.

References

Arbortext. 2000. Epic Editor 4.0 and Epic Editor 4.0 LE Online help. Internal tutorial, available with Epic Editor 4.0.

Arbortext. 2002. Epic Editor Overview [Web document]. Available: http://www.arbortext.com/html/epic_editor_overview.html. [Referenced 25.6.2002].

Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. & Wallnau, K. 2000. Volume II: Technical Concepts of Component-Based Software Engineering. Software Engineering Institute, report CMU/SEI-2000-TR-008.

Bass, L., Clements, P. & Kazman, R. 1998. Software architecture in practice. Reading, Massachusetts: Addison-Wesley. 452 p. ISBN 0-201-19930-0.

Bertoa, M. F. & Vallecillo, A. 2002. Quality Attributes for COTS Components. 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Málaga, Spain, June 2002. To be published by Springer-Verlag as a LNCS volume. 11 p.

Beus-Dukic, L. 2000. Non-Functional Requirements for COTS Software Components. ICSE 2000 Workshop on Continuing Collaborations for successful COTS Development, Limerick, Ireland, June 2000. Available: http://wwwsel.iit.nrc. ca/projects/cots/icse2000wkshp/positions.html. [Referenced 12.3.2002]. 5 p.

Biggs, W. & Evans, H. 2001. Simplify XML programming with JDOM [Web document]. Available: http://www-106.ibm.com/developerworks/library/j-jdom/. [Referenced 12.3.2002].

Bosch, J. 2000. Design and use of software architectures. Adopting and evolving a product-line approach. Harlow: Addison-Wesley. 354 p. ISBN: 0-201-67494-7.

Brownsword, L., Oberndorf, P. & Sledge, C. 2000. An Activity Framework for COTS-Based Systems. Software Engineering Institute, Technical Report CMU/SEI-2000-TR-010.

Clapp, J. A. & Taub, A. E. 1998. A Management Guide to Software Maintenance in COTS-Based Systems. Bedford, Massachusetts: MITRE, Center for Air Force C2 Systems. 32 p.

Clements, P. & Northrop, L. 2001. Software product lines. Practices and Patterns. New York: Addison-Wesley. 608 p. ISBN: 0201703327.

Component Registry Homepage 2002. Available: http://www.componentregistry.com/. [Referenced 26.3.2002].

ESA 1991. Guide to the user requirements definition phase. ESA Board for Software Standardisation and Control. ESA PSS-05-02 Issue 1. 34 p.

Flashline Inc. 2002. Available: http://www.flashline.com/. [Referenced 26.3.2002].

Forsell, M. & Päivärinta, T. 2002. A model for documenting reusable software components. In: Forsell, M. 2002. Improving Component Reuse in Software Development. Jyväskylä: Jyväskylä University Printing House. Pp. 117–150.

Garlan, D., Allen, R. & Ockerbloom, J. 1995. Architectural Mismatch or why it's hard to build systems out of existing parts. Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, April 1995. ACM Press, New York, USA. Pp. 179–185.

Harold, E. 1999. XML Bible. Foster City, USA: IDG Books WorldWide, Inc. 1015 p. ISBN: 0-7645-3236-7.

Harold, E. 2001. Processing XML with Java [Web document]. To be published by Addison-Wesley in November 2002. Available: http://www.ibiblio.org/xml/books/xmljava/. [Referenced 25.4.2002].

Heineman, G. T. & Councill, W. T. 2001. Component-Based Software Engineering. Putting the Pieces Together. New York: Addison-Wesley. 817 p. ISBN: 0-201-70485-4.

IEE. 1986. Guidelines for the documentation of software in industrial computer systems. London: The Institution of Electrical Engineers. ISBN: 086341 046 4.

IEEE Std 1062-1993. 1993. IEEE Recommended Practice for Software Acquisition. The IEEE Computer Society, Brussels, Belgium.

IEEE Std 610.12-1990. 1990. IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers, Inc., New York, USA.

iLogix. 2002. Rhapsody in C++ [Web document]. Available: http://www.ilogix.com/. [Referenced 18.2.2002].

Iribarne, L., Vallecillo, A., Alves, C. & Castro, C. 2001a. A Non-functional Approach for COTS Components Trading. Proceedings of the 4th Workshop on Requirements Engineering, Buenos Aires, Argentina, November 2001. Tercer Milenio, Buenos Aires, 2001. Pp. 124–138.

Iribarne, L., Troya, J. M. & Vallecillo, A. 2001b. Trading for COTS components in Open Environments. Proceedings of the 27th Euromicro Conference on Component-Based Software Engineering, Warsaw, Poland, September 2001. IEEE Computer Society Press. Pp. 22–29.

Iribarne, L., Vallecillo, A. & Troya, J. M. 2002. COTStrader. The trading process of COTS components [Web document]. Available: http://www.cotstrader.com/. [Referenced 25.3.2002].

ISO/IEC-9126. 1991. Information technology – Software product evaluation – Quality characteristics and guidelines for their use. International Standard ISO/IEC 9126, International Standard Organization, Geneve.

Jacobson, I., Booch, G. & Rumbaugh, J. 1998. The Unified Software Development Process. New York: Addison-Wesley. 463 p. ISBN: 0-201-57169-2.

JDOM Homepage. 2002. Available: http://www.jdom.org/. [Referenced 27.3.2002].

Kallio, P. & Niemelä, E. 2001. Documented Quality of COTS and OCM Components. Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto, Canada, May 2001. IEEE Computer Society, USA. Pp. 111–114.

Karlsson, E.-A. 1996. Software reuse – a holistic approach. Chichester: John Wiley & Sons. 510 p.

Kay, M. 2001. Anatomy of XSLT processor [Web document]. Available: http://www-106.ibm.com/developerworks/library/x-xslt2/. [Referenced 3.9.2001].

Kontio, J. 1996. A Case Study in Applying a Systematic Method for COTS Selection. Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, CA, USA. Pp. 201–209.

Kunda, D. & Brooks, L. 1999. Applying social-technical approach for COTS selection. Proceedings of 4th UKAIS Conference, University of York, April 1999. In: Brooks L., Kimble, C. 1999. Information Systems – The Next Generation. McGraw Hill. Pp. 552–565. ISBN 0 07 709558 8.

Matinlassi, M., Niemelä, E. & Dobrica, L. 2002. Quality-Driven Architecture Design and Quality Analysis Method. A revolutionary initiation approach to a product line architecture. Espoo: Technical Research Centre of Finland. VTT Publications 456. 128 p. + app. 10 p.

McClure, C. 1994. Reuse Engineering: Extending Information Engineering to Enable Software Reuse [Web document]. Extended Intelligence, Inc. Available: http://www.reusability.com/papers4.html. [Referenced 12.8.2002].

Meyers, B. C. & Oberndorf, P. 2001. Managing Software Acquisition: Open Systems and COTS Products. New York: Addison-Wesley. 360 p. ISBN: 0-201-70454-4.

Microsoft. 2002a. COM Interfaces [Web document]. Available: http://www.microsoft com/technet/treeview/default.asp?url=/TechNet/prodtechnol/host/proddocs/hisdoc/appint/ comti_com_interfaces.asp. [Referenced 15.4.2002].

Microsoft. 2002b. MSXML parser 3.0 [Web document]. Available: http://msdn.microsoft.com/downloads/. [Referenced 13.8.2001].

Morisio, M. & Sunderhaft, N. 2000. Commercial-Off-The-Shelf (COTS): A Survey. A DACS State-of-the-Art Report Contract Number SP0700-98-D-4000, Data Analysis Center for Software, Rome, NY. 95 p.

Ncube, C. & Maiden, N. 2000. COTS Software Selection: The Need to make Tradeoffs between System Requirements, Architectures and COTS/Components. ICSE 2000 Workshop on Continuing Collaborations for successful COTS Development, Limerick, Ireland, June 2000. Available: http://wwwsel.iit.nrc.ca/projects/cots/ icse2000wkshp/positions.html. [Referenced 15.4.2002]. 9 p.

Niemelä, E. 1999. A component framework of a distributed control systems family. Espoo: Technical Research Centre of Finland. VTT Publications 402. 188 p. + app. 68 p.

Niemelä, E., Kuikka, S., Vilkuna, K., Lampola, M., Ahonen, J., Forssel, M., Korhonen, R., Seppänen, V. & Ventä, O. 2000. Tekes-raportti: Teolliset komponentti-ohjelmistot. Kehittämistarpeet ja toimenpide-ehdotukset. Teknologiakatsaus 89/2000.

Niemelä, E., Taulavuori, A. & Kallio, P. 2002. Component Documentation – a Key Issue in Software Product Lines. Submitted to Journal of Systems and Software. 24 p.

Northrop, L. 2002. A Framework for Software Product Line Practice. Version 3.0. [Web-document]. Software Engineering Institute, Carnegie Mellon University, Pittsburgh. Available: http://www.sei.cmu.edu/plp/framework.html. [Referenced 22.1.2002].

NPLACE. 2002. The National Product Line Asset Center, NPLACE. Available: http://www.nplace.wvhtf.org/. [Referenced 22.5.2002].

Object Management Group. 2002. XML Metadata Interchange (XMI) [Web document]. Available: http://xml.coverpages.org/xmi.html. [Referenced 13.3.2002].

Ogush, M., Coleman, D. & Beringer, D. 2000. A template for documenting software and firmware architectures [Web-document]. Hewlett-Packard. Available: http://www.architecture.external.hp.com/index.htm. [Referenced 26.9.2002].

Open Source Initiative. 2002. The Open Source Definition, version 1.9 [Web document]. Available at: http://www.opensource.org/docs/definition.html. [Referenced 22.5.2002].

Park, R. 1992. Software Size Measurement: A Framework for Counting Source Statements, CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA.

Pressman, R. 1997. Software engineering. A practitioner's approach, 4th edition. New York: McGraw-Hill. 885 p. ISBN: 0 07 709 411 5.

Sametinger, J. 1997. Software engineering with reusable components. New York: Springer Verlag. 272 p. ISBN: 3-540-62695-6.

Seppänen, V., Helander, N., Niemelä, E. & Komi-Sirviö, S. 2001. Towards original software component manufacturing. Espoo: VTT. VTT Research Notes 2095. 105 p.

SER Consortium. 1996. Solutions for Software Evolution and Reuse III [Web document]. SER Esprit Project 9809. Available: http://dis.sema.es/projects/SER/solutions/solutions3.html. [Referenced 4.4.2002].

Sommerville, I. 1992. Software engineering, 4th edition. Workingham: Addison-Wesley. 649 p. ISBN: 0-201-56529-3.

Svahnberg, M. & Bosch, J. 2000. Issues Concerning Variability in Software Product Lines. Proceedings of the Third International Workshop on Software Architectures for Product Families, Gran Canaria, Spain, March 2000. Heidelberg, Germany: Springer LNCS. Pp. 146–157.

Taulavuori, A., Kallio, P. & Niemelä, E. 2002. Documentation system of commercial components. Accepted to the 15th International Conference on Software & Systems Engineering, ICSSEA 2002, Paris, December 2002. 9 p.

Vidger, M. & Dean, J. 1997. An Architectural Approach to Building Systems from COTS Software Components. Proceedings of the 22nd Annual Software Engineering Workshop, Greenbelt, Maryland, December 1997. Technical Report 40221, National Research Council.

Vigder, M. 1998. An Architecture for COTS Based Software Systems. NRC Report No. 41603. Canada: National Research Council of Canada. 22 p.

Wallace, D. R., Peng, W. W. & Ippolito, L. M. 1992. Software Quality Assurance: Documentation and Reviews [Web document]. NIST IR 4909. Available: http://hissa.ncsl.nist.gov/publications/nistir4909/. [Referenced 17.6.2002].

Walsh, N. 1998. A Technical Introduction to XML. InterCHANGE, Vol. 4, Issue 2. Pp. 17–26.

World Wide Web Consortium. 2001a. Extensible Markup Language (XML) [Web document]. Available: http://www.w3.org/XML. [Referenced 28.8.2001].

World Wide Web Consortium. 2001b. XML Schema [Web document]. Available: http://www.w3.org/XML/Schema. [Referenced 28.8.2001].

World Wide Web Consortium. 2002a. Document Object Model (DOM) [Web document]. Available: http://www.w3c.org/DOM. [Referenced 15.2.2002].

World Wide Web Consortium. 2002b. The Extensible Stylesheet Language [Web document]. Available: http://www.w3c.org/Style/XSL. [Referenced 28.8.2001].

Yacoub, S., Mili, A., Kaveri, C. & Dehlin, M. 2000. A Hierarchy of COTS Certification Criteria. Proceedings of the First Software Product Line Conference, SPLC1, Denver, Colorado, August 2000. Kluwer Academic Publishers, Boston. Pp. 397–412.

Yakimovich, D., Bieman, J. M. & Basili, V. R. 1999. Software architecture classification for estimating the cost of COTS integration. Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, May 1999. IEEE Computer Society Press, Los Alamitos, CA, USA. Pp. 296–302.

Appendix A: Questions for the interviews about the component documentation

General questions:

What information should the (third party) component documentation include?

How specific should the documentation be?

Does the product line define special requirements for documentation?

How should the quality of the component be documented?

What are the requirements for component documentation?

What should the structure of the documentation be?

What is the maximum length of the documentation?

In which phases of the component development should the information for the documentation be developed?

Who should develop each part of the documentation?

Who uses the documentation information and for what purposes?

What information should the component buyer receive before purchasing the component?

Questions about OCM components:

Does the documentation of OCM components differ from the documentation of COTS components? How?

How should the buyer prepare the requirements specification of an OCM component?

Is the documentation of the architecture of an OCM component necessary in component documentation?

Component documentation in the interviewee's own company:

Does your company have regulations for component documentation?

How are the components documented in your company today?

What is the purpose of the documentation in your company?

Who is responsible for documentation in your company?

Component search and selection:

How does your company make the decision whether to search for a COTS component, reuse an internal component, develop a new component or order an OCM component for a special purpose? Who does this decision in your company?

What does the component requirements specification include?

Is the requirements specification the same for a new component and an existing component?

Is the functionality of the component defined specifically enough in your company for the search? How is the functionality defined?

How are the quality requirements defined in your company?

What are the common problems in component search and selection?

How could documentation ease the selection of components?

Can one ever be sure that the third-party component is applicable to the architecture in which it is intended to be used?

How could the applicability of a component to the product line architecture be assured?

Does the delivery time of the component have an impact on component selection?

Integration of the component:

With what information can the success of the integration be guaranteed?

What are the problems in component integration?

How could documentation solve these problems?

How can one be sure that the component can be integrated using the integration components? How should the integration components be documented?

Should the interface code of the component be included in the component documentation?

What information enables the modification of a component (OCM component)?

Testing of the component:

Is the testing information required in component documentation?

How extensively should the component testing be documented?

How could the testing information of the component be associated with the integration testing of the product?

Questions about the developed documentation pattern:

Thoughts about the pattern?

Improvement suggestions?

Is the pattern adequate for component selection, integration and use?

Published by

Series title, number and report code of publication

VTT Publications 484
VTT–PUBS–484

Author(s)
Taulavuori, Anne

Title

Component documentation in the context of software product lines

Abstract

The use of third-party components in software system development is rapidly increasing. The product lines have also adopted this new tendency, as the COTS and OCM components are increasingly being used in product-line-based software engineering. Component documentation has become a key issue in component trading because it often is the only way of assessing the applicability, credibility and quality of a third-party component, especially for product lines in which the common architecture determines the decisive requirements and restrictions for components. However, at the present time there is no standard model for component documentation, and, therefore, the component documents are often inconsistent, insufficient and of various quality. The lack of a standard documentation model is thus one of the bottlenecks in component trading.

The purpose of this thesis is to define the documentation requirements of software components and form a standard documentation pattern from these requirements. The documentation requirements are examined from the viewpoint of the software product lines, where the common product line architecture may define several specific requirements for a component. The standard pattern is a skeleton of documentation, defining the content and structure for component documentation. The pattern ensures the documentation that assists the integrator in successful component selection, validation, integration and use within product lines. The development of the documentation is defined by identifying the roles responsible for the documentation and associating them with the pattern.

Definition of the documentation pattern is not sufficient for the adoption of a new documentation practice. An environment that supports the development of documentation is also required. This thesis also introduces the developed documentation system, which defines how the component documentation could be implemented. The system provides guidelines concerning how to document a software component. It also offers the tools and technology for the development and handling of documents, and ensures that the developed documentation is in accordance with the pattern. In addition, the system is also applicable when the development of the documentation is split between different organisations. An evaluation of the documentation pattern is presented at the end of this thesis.

Keywords
component documentation, software product lines, software engineering, component documentation pattern

Activity unit
VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN			Project number
951–38–6021–3 (soft back ed.) 951–38–6022–1 (URL:http://www.inf.vtt.fi/pdf/)			E1SU00321
Date December 2002	Language English	Pages 111 p. + app. 3 p.	Price C
Name of project Minttu		Commissioned by	
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

VTT PUBLICATIONS

466 Vasara, Tuija. Functional analysis of the RHOIII and 14-3-3 proteins of *Trichoderma reesei*. 93 p. + app. 54 p.

467 Tala, Tuomas. Transport Barrier and Current Profile Studies on the JET Tokamak. 2002. 71 p. + app. 95 p.

468 Sneck, Timo. Hypoteeseista ja skenaarioista kohti yhteiskäyttäjien ennakoivia ohjantajärjestelmiä. Ennakointityön toiminnallinen hyödyntäminen. 2002. 259 s. + liitt. 28 s.

469 Sulankivi, Kristiina, Lakka, Antti & Luedke, Mary. Projektin hallinta sähköisen tiedonsiirron ympäristössä. 2002. 162 s. + liiitt. 1 s.

471 Tuomaala, Pekka. Implementation and evaluation of air flow and heat transfer routines for building simulation tools. 2002. 45 p. + app. 52 p.

472 Kinnunen, Petri. Electrochemical characterisation and modelling of passive films on Ni- and Fe-based alloys. 2002. 71 p. + app. 122 p

473 Myllärinen, Päivi. Starches – from granules to novel applications. 2002. 63 p. + app. 60 p.

474 Taskinen, Tapani. Measuring change management in manufacturing process. A measurement method for simulation-game-based process development. 254 p. + app. 29 p.

475 Koivu, Tapio. Toimintamalli rakennusprosessin parantamiseksi. 2002. 174 s. + liitt. 32 s.

477 Purhonen, Anu. Quality driven multimode DSP software architecture development. 2002. 150 p.

478 Abrahamsson, Pekka, Salo, Outi, Ronkainen, Jussi & Warsta, Juhani. Agile software development methods. Review and analysis. 2002. 107 p.

479 Karhela, Tommi. A Software Architecture for Configuration and Usage of Process Simulation Models. Software Component Technology and XML-based Approach. 2002. 129 p. + app. 19 p.

480 Laitehygienia elintarviketeollisuudessa. Hygieniaongelmien ja *Listeria monocytogeneksen* hallintakeinot. Gun Wirtanen (toim.). 2002. 183 s.

481 Wirtanen, Gun, Langsrud, Solveig, Salo, Satu, Olofson, Ulla, Alnås, Harriet, Neuman, Monika, Homleid, Jens Petter & Mattila-Sandholm, Tiina. Evaluation of sanitation procedures for use in dairies. 2002. 96 p. + app. 43 p.

482 Wirtanen, Gun, Pahkala, Satu, Miettinen, Hanna, Enbom, Seppo & Vanne, Liisa. Clean air solutions in food processing. 2002. 93 p.

483 Heikinheimo, Lea. *Trichoderma reesei* cellulases in processing of cotton. 2002. 77 p. + app. 37 p.

484 Taulavuori, Anne. Component documentation in the context of software product lines. 2002. 111 p. + app. 3 p.

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU PL 2000 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	VTT INFORMATIONSTJÄNST PB 2000 02044 VTT Tel. (09) 456 4404 Fax (09) 456 4374	VTT INFORMATION SERVICE P.O.Box 2000 FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

ISBN 951-38-6021-3 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 951-38-6022-1 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455-0849 (URL: http://www.inf.vtt.fi/pdf/)