NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK

INSTITUTT FOR DATATEKNIKK OG INFORMASJONSVITENSKAP

FORDYPNINGSPROSJEKT

Studenter:	Henrik Schwarz, Ole Morten Killi og Stein-Roar Skånhaug
Fag:	SIF8094 – Fordypningsprosjekt, systemutvikling
Tittel:	A Study of Industrial, Component-Based Development, Ericsson

Oppgavetekst:
The object of this assignment is to study the state-of-the-art of development processes with focus on component identification and state-of-the-art standards for component specification. This understanding should be applied to the case from the Ericsson GPRS project and the outcome is analysis of the current practice and possibly proposals for improvement.

Oppgaven gitt:	28. august 2002
Besvarelsen leveres innen:	22. november 2002
Utført ved:	IDI / NTNU, Gløshaugen, Trondheim
Veileder (intern):	Reidar Conradi, NTNU
Veileder (ekstern):	Parastoo Mohagheghi, Ericsson

Abstract

Title: A study of the state-of-the-art of development processes with focus on component identification and standards for component specification in large scale systems.

Background: Components like objects follow the principle of combining functions and data into a single unit. While several proposals are made on how to identify objects (or classes) such as a grammatical analysis of system description (nouns-> objects, verbs-> services, etc) or scenario based analysis, there is little guidance for identifying components and their interfaces. Having in mind that large-scale systems are a collection of components, identification of components from functional and non-functional requirements is the goal of the analysis and design process.

The GPRS project in Ericsson is a large, real-time system where an adaptation of the Rational Unified Process (GSN RUP) is chosen as the software development process. This process also lacks definition of tasks or activities for identification of components in the analysis and design phase. As RUP uses earlier versions of UML for modeling, the notation of components as physical identities in deployment view is still dominant. However UML 2.0 is taking the step towards defining components earlier in the design process.

Definition: The object of this assignment is to study the state-of-the-art of development processes with focus on component identification and state-of-the-art standards for component specification. This understanding should be applied to the case from the Ericsson GPRS project and the outcome is analysis of the current practice and possibly proposals for improvement.

Tasks:

- Study state-of-the-art of development processes with focus on component identification
- Study state-of-the-art standards for component specification.
- Study the current practice at Ericsson regarding component identification in GSN RUP.
- Study the current practice at Ericsson regarding component specification.
- Propose improvements for the current practice at Ericsson.

Preface

This report is the result of work performed in the course "SIF8094 Fordypningsprosjekt, systemutvikling" during the autumn of 2002. The course is a part of the ninth semester of the Master of Science study at NTNU.

This project is a part of the larger INCO project, which is a research project on incremental and component-based development funded by NFR (Norges forskningsråd).

We wish to thank our project advisors that have made it possible for us to achieve our goals in this project. First we want to thank our project supervisor, Professor Reidar Conradi at NTNU, for providing insightful input and valuable feedback throughout the project. Second we wish to thank our external contact and project advisor, Parastoo Mohagheghi at Ericsson AS, for guiding us in our study of the current practice at Ericsson and providing valuable feedback on the report.

Trondheim, November 22, 2002

Henrik Schwarz Ole Morten Killi Stein-Roar Skånhaug

Table of Contents

Abstract 2
Preface 3
Table of Contents 4
Table of Figures 6
1 Introduction 8
1.1 Purpose 8
1.2 Motivation 8
1.3 Context 8
1.4 Intended Audience 8
1.5 About Report Structure 9
1.6 Reading Guide 9
2 Survey of State-of-the-Art 10
2.1 Development Processes 10
2.1.1 Component-Based Software Engineering (CBSE) 10
2.1.2 Rational Unified Process (RUP) 13
2.1.3 KobrA 17
2.1.4 COMET 26
2.2 Standards 33
2.2.1 Unified Modeling Language (UML) 33
2.2.2 CORBA Component Model (CCM) 42
2.2.3 Reusable Asset Specification (RAS) 48
2.3 Evaluation of State-of-the-art 61
2.3.1 Comparison of Processes 61
2.3.2 Comparison of Standards 63
2.3.3 Summary 63
3 Current Practice at Ericsson 64
3.1 About Ericsson 64
3.2 Ericsson GSN 64
3.3 GSN RUP Adaptation 65
3.3.1 GSN RUP Overview 65
3.3.2 Requirements Workflow 66
3.3.3 Analysis & Design Workflow 74
3.3.4 Component Identification in GSN RUP 82
3.3.5 Product Line Engineering 83
3.4 GSN Architecture 84
3.4.1 The Core Network 84
3.4.2 GSN System Family 85
3.4.3 Definition of Architecture 85
3.4.4 Architectural Views 85
3.4.5 Application Platform and Framework 86
3.5 GSN UML Model 88
3.5.1 Logical View 88
3.5.2 Component View 92
3.5.3 Summary 92
3.6 Evaluation of Current Practice 93
3.6.1 GSN RUP 93
3.6.2 GSN UML Model 93
4 Improvement Suggestions 94
4.1 Introduction 94
4.2 Product Line Engineering in GSN RUP 94

4.3 Initial Component Identification in GSN RUP 94
4.4 Further Component Specification in the GSN Model 95
4.5 Summary 97
5 Conclusion and Further Work 98
6 References and Abbreviations 99
Appendix A: Details on RAS Integration in RUP 102

Table of Figures

Figure 2-1: Traditional software reuse........11
Figure 2-2: RUP phases and workflows........13
Figure 2-3: KobrA component (Komponent) specification and realization........19
Figure 2-4: Notation in figures........21
Figure 2-5: Product line engineering in KobrA........21
Figure 2-6: Framework engineering in KobrA........22
Figure 2-7: COMET models and their overall relations........27
Figure 2-8: COMET process overview modeled with UML activity diagram........29
Figure 2-9: How COMET supports evolutionary development and reuse........31
Figure 2-10: Package Diagram example........34
Figure 2-11: Deployment Diagram containing Component Diagrams........34
Figure 2-12: Component realizing a component specification........35
Figure 2-13: Component offering interface types........35
Figure 2-14: Component specification diagram........36
Figure 2-15: A Component with one provided interface.........38
Figure 2-16: A Component with two provided and three required interfaces.........38
Figure 2-17: Alternative black box notation of a component.........39
Figure 2-18: Component diagram showing a white-box view of a ‘nested’ component........39
Figure 2-19: Components and their general dependencies........40
Figure 2-20: Example of a component deployment diagram........40
Figure 2-21: Sequence Diagram with reference........41
Figure 2-22: Traditional CORBA Object Model........42
Figure 2-23: Example of a CCM component with corresponding IDL code........43
Figure 2-24: CCM component interaction through port mechanisms........45
Figure 2-25: Classification of assets........49
Figure 2-26: Logical Asset Structure Domain Model........50
Figure 2-27: Classification section model........51
Figure 2-28: Solution model........52
Figure 2-29: Usage Model........53
Figure 2-30: Assets Relationship........54
Figure 2-31: Asset Layout........55
Figure 2-32 - Aladdin Reuse Portal, screen shot........56
Figure 2-33: Inserting an asset into the Rose workspace........57
Figure 2-34: Rational Asset Browser standalone tool........58
Figure 2-35: RAMP Architecture........59
Figure 2-36: Comparison chart over KobrA, COMET and RUP........61
Figure 2-37: Boehms planning spectrum........62
Figure 2-38: Proposed placement of methods in Boehms planning spectrum........62
Figure 2-39: Comparison chart over UML 1.x, UML 2.0 and CCM........63
Figure 3-1: GSN RUP process map........65
Figure 3-2: Overview of the Requirement Workflow........66
Figure 3-3: Overview of the workflow detail "Understand stakeholders’ needs"........67
Figure 3-4: Overview of the workflow detail "Define the System"........68
Figure 3-5: Proposed overview of the workflow detail “Develop Feature Impact Study”........70
Figure 3-6: Overview of the workflow detail "Manage Changing Requirements"........71
Figure 3-7: Overview of the workflow detail "Manage the Scope”........72
Figure 3-8: Overview of the workflow detail "Refine System Definition"........73
Figure 3-9: Analysis & Design workflow........74
Figure 3-10: Overview of the workflow detail "System Architecture Analysis Workflow"........75
Figure 3-11: Overview of the workflow detail "Architecture Refinement Analysis"........77

Figure 3-12: Overview of the workflow detail "Design Input Analysis" 77
Figure 3-13: Overview of the workflow detail "Behavioral Analysis" 78
Figure 3-14: Overview of the workflow detail "System Behavioral Design"............................79
Figure 3-15: Overview of the workflow detail "Component Software Design"........................80
Figure 3-16: GSN component types and their interfaces 81
Figure 3-17: Product Line Engineering in GSN RUP as we see it 83
Figure 3-18: Overview of the Ericsson packet-data core network...84
Figure 3-19: The GSN System family...85
Figure 3-20: High level packages and their dependencies 86
Figure 3-21: Layered architecture 87
Figure 3-22: Different types of design elements that are used in the design model 89
Figure 3-23: Example of a unit dependency diagram...89
Figure 3-24: Subsystem class diagram example 90
Figure 3-25: Interface example 91
Figure 3-26: Component diagram example..92
Figure 4-1: Overview of the problem areas identified in GSN 94
Figure 4-2: Proposed timeline for introducing our improvements at Ericsson........................97

1 Introduction

1.1 Purpose

This report is a study of state-of-the-art processes and standards for software development. We also study the current practice at Ericsson in order to find improvement potentials on their adapted RUP process. We then use the explored processes and standards to propose improvements.

1.2 Motivation

Component-based software development offers great advantages to development organizations in the form of lowered costs, higher software quality and shorter time-to-market. Component reuse is a fairly new field in the history of software engineering, and there is still much to explore. Recently there have been introduced new development processes and modeling standards that address component specification. We wanted to study these processes and standards and how Ericsson could benefit from these. By learning how GSN RUP identifies components in the requirement and analysis phase and how GSN models components in UML 1.x we hoped to be able propose solutions on how new processes like KobrA / COMET and standards like UML 2.0, CCM and RAS could be useful for Ericsson.

1.3 Context

This report is a part of the Incremental and Component-based development (INCO) research project. The INCO project is a joint project between the Department of Computer and Information Science (IDI) at NTNU and the Department of Informatics (IFI) at the University in Oslo (UiO), supported by The Norwegian Research Council (NFR). Software development today is highly costly and there is always a great risk for failure. Recently it has been purposed that incremental and component-based development will reduce these problems. These techniques are immature in the form of missing industrial support. The INCO project will explore and gather experience of these challenges. Our study will explore techniques for identification and specification of components and how Ericsson's GSN project could benefit from these.

1.4 Intended Audience

This report is aimed at any software designers or architects who want to get an overview of the state-of-the-art of development process with focus on component identification, an introduction to the state-of-the-art standards for component specification and how Ericsson does component-based development in the GSN project.

Readers of this report should at least have basic knowledge of UML 1.x, the Rational Unified Process and general knowledge in software engineering.

1.5 About Report Structure

This report is composed of three main parts. Chapter 2 is a study on the state-of-the-art of development processes and standards for component specification. Chapter 3 is a study of the current practice on Ericsson's GSN project. The last part is our improvement suggestions on current practice at Ericsson.

We have chosen to treat the topics of processes and standards separately in order not to mix the two.

1.6 Reading Guide

This report can be read in several ways.

If you are interested in state-of-the-art development processes (RUP, KobrA and COMET) and standards for component specification (UML, CCM and RAS), read chapter 2. This chapter can be skipped if you already are familiar with these processes and standards. In any case we recommend you to read chapter 2.3, Evaluation of State-of-the-art, for a comparison of the mentioned processes and standards.

If you want to read about the current practice at Ericsson, component identification and specification in GSN RUP, read chapter 3. This chapter also contains a description of the GSN UML model. Skip this chapter if you already are familiar with the current practice at Ericsson.

If you just are interested in our suggested improvements on the current practice at Ericsson, read chapter 4.

2 Survey of State-of-the-Art

We have chosen to split this chapter into two main parts: The first part contains an introduction to Component-Based Software Engineering (CBSE) followed by a study of the state-of-the-art of development *processes* (RUP, KobrA and COMET) with focus on component identification. The second part of this chapter contains a study of the state-of-the-art of *standards* for component specification (UML 2.0 and CCM) and asset specification (RAS).

2.1 Development Processes

This chapter begins with an introduction to Component-Based Software Engineering (CBSE) and the Rational Unified Process (RUP). We then study the state-of-the-art of development processes with focus on component identification, including KobrA and COMET.

2.1.1 Component-Based Software Engineering (CBSE)

This chapter contains information from [Atkinson02], [Solberg97] and [Poulin95].

The need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software. CBSE is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usages. Developing systems out of existing components offers many advantages to developers and users. In component-based systems:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- System evolution is eased because system built on CBSE concepts is open to changes and extensions, i.e., components with new functionality can be plugged into an existing system.
- Quality of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and changes in a component are, therefore, beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user. Component-based development also raises many challenging problems, such as:

- Building good reusable components. This is not an easy task and a significant effort must be used to produce a component that can be used in different software systems.
- In particular, components must be tested and verified to be eligible for reuse.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly.

- The same problem arises when a new component needs to be integrated into an existing system.
- Verification of reusable components. Components are developed to be reused in many different systems, which make the component verification a significant challenge.
- For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.
- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

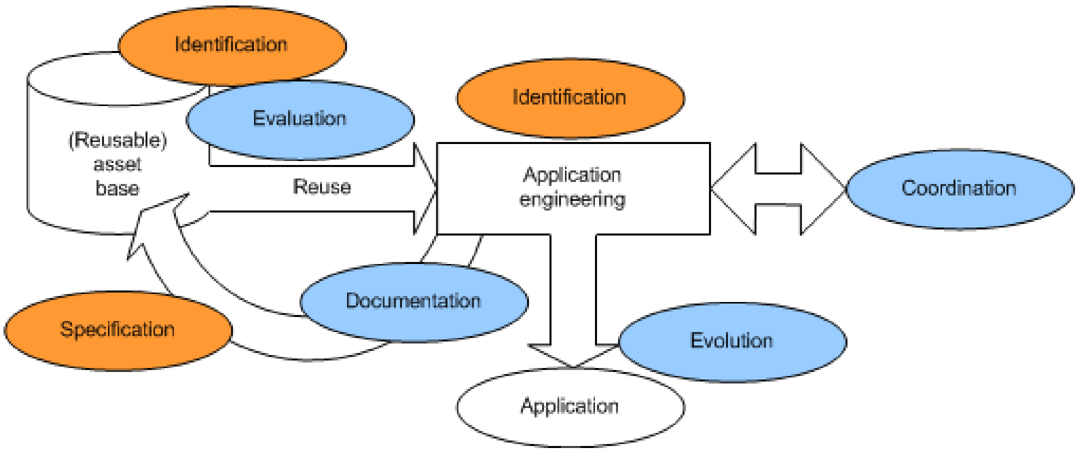


Figure 2-1: Traditional software reuse

Figure 2-1 (from [Atkinson02]) illustrates the traditional “reuse” life-cycle based on software artifacts (i.e. components) initially developed for use with one application in mind. Basically, the artifacts created in one application-engineering project are stored in an asset base so that they can be reused in other applications. Future application projects then attempt to develop parts of the application by reusing existing assets in the asset base (/reuse library). This kind of reuse is usually white-box reuse where source code is available.

The circles marked in blue and orange (in figure 2-1) are typical problem areas that have to be addressed for a successful reuse approach. In this study we focus on the identification and definition part (marked in orange). Identification covers both deciding what components that should be developed from the requirements and the identification of reusable components. Specification covers how components are specified. In addition to these two aspects, the economic side of attempting to create reusable components for the future (making more generalized components usually takes more time) should be considered. The engineers should inform the customer what kind of impact the requirements (and the costs that follows it) have on his/her choices for the software. Involving the customer in this process might cause requirements to change, allowing a reusable component to be used. This is important to do as early as possibly as the requirements are often more easily changeable early on in the project. Typically, a reusable component “earns” its value after 2-4 years, by being reused at least twice. In addition the net economic savings are typically 20-80% return on initial investments according to [Sjøberg00]. Even with this perspective developing for reuse might not be profitable because the reusable components created might not be reused at all. Ralph Johnson has emphasized that reusable components are “not (pre)planned, they are discovered (gradually later)” [Gamma95] and Jeffrey Poulin presents three phases of the corporate reuse libraries [Poulin95]:

1. very few parts,
2. many parts of low or poor quality,
3. many parts of little or no use.

Johnson's and Poulin's remarks should be kept in mind to prevent too many components to be developed for reuse that are not reused later on. It is also important not to overestimate the usefulness of an asset base and that an asset base requires extra documentation for each asset to offer effective searches in the base.

Thinking reuse when developing is the key for taking full advantage of CBSE possibilities. CBSE covers both reuse of traditional libraries and object-oriented frameworks, as well as program/product families and COTS (Commercial-Of-The-Shelf). COTS-based reuse has proved hard [Garland95] [Carney97] [Boehm99] [Morisio00] because there is no standardized development process for doing this. In addition we have no control over COTS risk factors, such as incompatible new versions, vendor delays and inadequate documentation.

2.1.2 Rational Unified Process (RUP)

This chapter contains information from [Kruchten00] and [Naalsund01].

The Rational Unified Process (RUP) is a software engineering process, delivered through a web-enabled, searchable knowledge base. The goal of the process is to produce, within a predictable schedule and budget, high-quality software that meets the needs of its end-users. By providing each team members with easy access to a knowledge base with guidelines, templates and tool mentors, it is meant to enhance the team productivity. This is to ensure a common language, process and view of how to develop software among members working with different phases of the project. RUP creates and maintains models, as well as being a guide for how to effectively use the Unified Modeling Language (UML). It is also a configurable process that can be adapted and extended to suit the need of an adopting organization.

RUP properties:

- Iterative development
- Use of component-based architecture
- Requirement management
- Software modeling done visually (UML)
- Software quality verification
- Software change control

2.1.2.1 Methodology Overview

A workflow is a sequence of activities that produces a result of observable value. The vertical axis of shows the six core engineering workflows and the three core supporting workflows. The horizontal axis represents the phases that the process loops trough during its lifecycle.

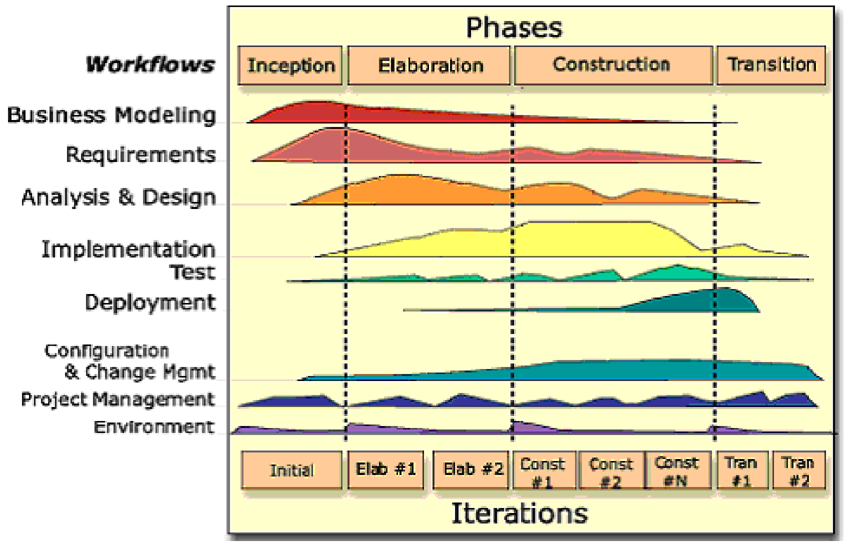


Figure 2-2: RUP phases and workflows

The Inception Phase

In this phase, the company needs to establish a business case for the system and delimit the project scope. All external actors must be identified, and the interaction with these must be defined. In the business case, the resources needed for the project is estimated. A phase plan, showing dates of major milestones is also included, along with success criteria and risk assessments.

The outcome of the inception phase covers a project vision (a vision of the core requirements of the project, key features and main constraints), an initial use-case model, an initial business case (with risk assessment, project phase plan and a business model) and a prototype.

The Elaboration Phase

In the elaboration phase, the purpose is to analyze the problem domain, establish an architectural foundation, develop the project plan, and eliminate the highest risk factors of the project. In order to accomplish these objectives, one must have a broad view of the project. Not too detailed knowledge, but a general understanding of the scope, major functionality and non-functional requirements such as performance requirements.

The outcome of this phase is a more complete use case model than earlier, supplemental requirements, a software architecture description, an executable prototype, a revised business case (with risk list), a development plan for the overall project (replacing and revising the initial project plan) and an updated development case specifying the process to be used.

The Construction Phase

In this phase, all remaining components and application features are developed and integrated into the product, and all features thoroughly tested. This phase is a manufacturing process with emphasis on managing resources and controlling operations to optimize costs, schedules, and quality.

The outcome of the construction phase is a product ready for the end-users. At a minimum, it consists of the software product integrated on the adequate platforms, the user manuals and a description of the current release.

The Transition Phase

This phase's purpose is to transition the software product to the user community. Once a product is released, issues arise that require the company to develop new releases, correct errors or finish the features that were postponed. The phase is entered when a baseline is mature enough to be deployed in the end-user domain.

The transition phase focuses on the activities required to place the software into the hands of the users. Typically, it requires several iterations, including beta releases, general availability releases, as well as bug fix and enhancement releases. A lot of effort is put in helping the user learn the product by developing user documentation and training users.

2.1.2.2 Component Definition

[Kruchten00] RUP definition of Component: *A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.*

RUP uses several perspectives on components:

- Development components (seen from the development organization point of view). This can be reusable subsystems that have low external coupling.
- Business components: Set of runtime components that fulfills a large chunk of business-level functionality.
- Runtime components: Components that are delivered, installed and run (i.e. executables and DLL's).

Subsystems in RUP are independent units of functionality that allow different parts of the system to be implemented independently. Subsystems are used in the logical view of RUP. The term "component" is only used in the implementation and deployment view of RUP but not in the logical view; hence components are not logical entities in the analysis & design phase of RUP. Refer to [Kruchten00] for a description of the different views of RUP.

2.1.2.3 Component Identification

[Kruchten00] In the Analysis & Design workflow the architecture is defined. In the "Define a Candidate Architecture" workflow detail the analysis classes are identified from the architecturally significant use cases. Then in the "Refine the Architecture" workflow detail the appropriate components (design elements) are identified from the analysis elements and integrated with preexisting design elements. The purpose of this workflow detail is also to ensure that maximal reuse of available components is achieved as early as possible in the design effort.

RUP is not very prescriptive on the process of component identification. RUP does not have activities on the identification of already existing components (design *with* reuse). This process would include searching for adequate COTS or internally developed components, and evaluating the quality and the appropriateness of candidate components. Also RUP lacks a process of defining to what extent a component is developed *for* reuse. When developing *for* reuse it is important to analyze and understand what requirements possible re-users may have as well as what cost benefits later reuses of the component may have. Proper research/analysis will have to be done in order to determine how general the component should be. [Naalsund01] proposes new activities for the adapted RUP process at Ericsson (GSN RUP) to improve component identification and reuse.

2.1.2.4 Summary

[Atkinson02] The RUP is clearly an extremely feature-rich method that embraces almost all the generally recognized techniques and artifacts of modern software engineering, including use-cases, architectures, patterns and components. Great emphasis is placed on elaborating these artifacts in an iterative and incremental manner. In other words, successive iterations gradually add more and more detail to the model until a final executable product is attained. This richness is both a strength and weakness of RUP. It embraces all the modern best practices of software engineering, but gives little concrete guidelines on how to resolve the tensions that inevitably arise. For example, the process is simultaneously “use-case driven” (i.e. function-oriented) and “architecture centric” (i.e. object-oriented), but it is not always clear which should take precedence when conflicts arise. Also, although the method divides up the work to be performed into small iterations as illustrated in, many of the iterations ignore the key process steps (or “workflows” as they are termed in RUP). Detailed implementation activities, such as coding and testing therefore still do not occur until the analysis and design activities have been performed on a system-wide basis. The basic phases of the waterfall model are therefore lurking in the background and still have a strong influence on the organization of products and process steps.

Not very prescriptive about the use of UML, even though it is intended to be the standard for UML development. As cited in [Atkinson02]: *It requires the creation of various models, but is not specific about which diagrams should be used to represent them and what precisely they should contain. In particular, the role of components is not entirely clear. As an artifact, a component is defined as “the physical packaging of model elements,” in line with the UML viewpoint. As a workflow, however, “component engineering” deals with all kinds of models at all abstraction levels, suggesting that a component is more of an architectural unit rather than a physical executable*).

Advantages:

- Uses iterative development
- Supports component-based development
- Visual modeling using UML
- Tool support
- Provides a description of both managerial and technical aspects
- Based on best-practices in software engineering field
- Well known process, wide-spread use in industry and adopted by major players

Disadvantages:

- Lack of activities that support component identification and reuse
- Components are not logical entities in the analysis & design workflow
- Not very prescriptive about the use of UML

2.1.3 KobrA

The information in this chapter is collected from [Atkinson00], [Atkinson01] and [Atkinson02]. The introduction below is from [Atkinson00].

The KobrA approach, developed in the KobrA project by Softlab GmbH, Psipenta GmbH, GMD-FIRST and Fraunhofer IESE, addresses the problem that to date the component paradigm has only really penetrated the "implementation" phase of the software life-cycle, and does not yet play a major role in the earlier analysis and design activities of large software projects. It does that by making components the focus of the entire software development process, not just the implementation and deployment phases, and by adopting a product-line strategy for their creation, maintenance and deployment. The resulting method augments the typical "binary-module" view of components with a full, UML-based representation that captures their entire set of characteristics and relationships. This not only makes the analysis and design activities component-oriented, but allows the essential structure and behavior of component-based systems to be described in a way that is independent of (but compatible with) specific component implementation technologies such as COM, CORBA or Java Beans.

2.1.3.1 Methodology Overview

Both product-line and component-based approaches to software development represent powerful techniques for supporting reuse. Components provide a technology for "reuse in the small" while product line development represents an approach for "reuse in the large". Therefore, significant benefits can be expected from their integration. The KobrA method attempts to create a natural synergy between the component-based and product line approaches to software development.

As far as possible there are no global or system-wide products - all products (and accompanying processes) are defined to carry information only related to their particular component. The advantage is that components (and the products that describe them) can then easily be separated from the environment in which they were developed and therefore can be reused independently.

In KobrA are components called Komponents.

2.1.3.2 Component Definition

The components in a KobrA framework or application are not physical components in the sense of contemporary “physical” component technologies (i.e. CORBA, .NET/COM+, J2EE/EJB), but rather “logical” components that represent the logical building blocks of a software system. This is closely connected to one of the central goals of KobrA; to enable the full expressive power of the UML to be used in the modeling of components. This is done by laying out four basic principles described in [Atkinson02]:

- Uniformity: every behavior-rich entity is treated as a Komponent, the system as a whole is viewed and modeled as a component)

- Encapsulation: the description of what a software unit does is separated from the description of how it does it)

- Locality: all descriptive artifacts represent the properties of a Komponent from a local perspective rather than a global perspective)

- Parsimony: every descriptive artifact should have “just enough” information, no more and no less)

Komponents are defined as: “*A logical component documented according to the KobrA approach. In particular, a Komponent has a specification, describing what the Komponent does, a realization, defining how it does it, and an implementation, describing how it is implemented using language-level constructs and physical components.*”

At the development time, software engineers are concerned with the description of Komponent types that capture the common properties of their run time Komponent instances. Like UML (described in 2.2.1), KobrA uses the unqualified term “component” to mean a component type. Komponents have both the properties of classes and the properties of modules (i.e. packages in UML terminology). The class-like facet allows a Komponent to define the attributes, operations and behavior possessed by its instances, while the module-like facet enables a Komponent to represent a name space and act as a container for features, classes, and other Komponents. In this respect Komponents are somewhat like subsystems in UML. Like a subsystem, a Komponent defines a grouping of model elements that represent a behavioral unit of a system. The difference between UML subsystems and Komponents is that a subsystem cannot possess behavior of its own, whereas Komponents can.

In KobrA, a framework is the static representation of a set of Komponents organized in the form of a tree. Each Komponent is described at two levels of abstraction - a specification, which defines the component's externally visible properties and behaviors, and thus serves to capture the contract that the Komponent fulfils, and a realization, which describes how the component fulfils this contract in terms of contracts with lower level components. A framework, therefore, is a tightly coupled arrangement of Komponent specifications and realizations. Figure 2-3 shows the general set of UML models, which make up Komponent specifications and realizations.

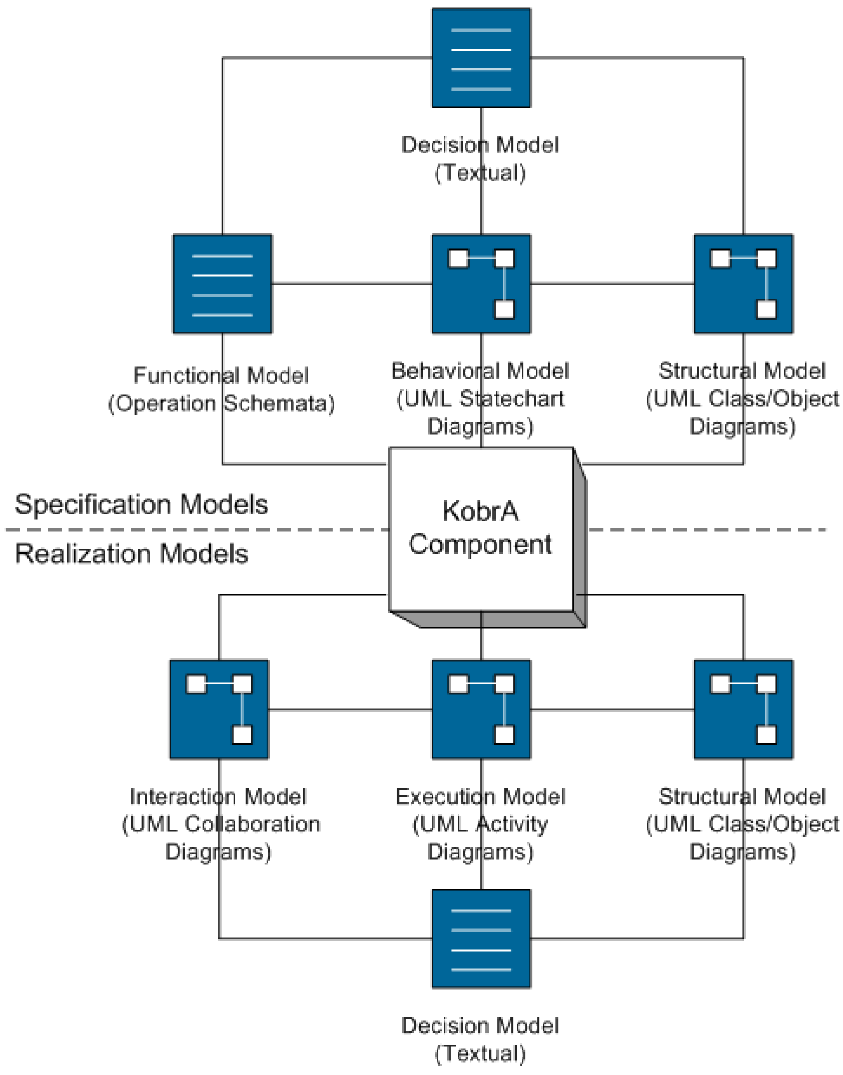


Figure 2-3: KobrA component (Komponent) specification and realization

The specification of a Komponent is comprised of four closely inter-related main models: the structural model, the behavioral model, the functional model, and the decision model. The structural, behavioral and functional models constitute the specification models for a Komponent as it is used in all applications covered by the framework. As seen in figure 2-3, the Komponent realization phase specifies an additional two models. The structural model and the decision model are found in both phases. According to [Atkinson02] the structural model from specification should be without the object diagrams (although this is not indicated in the figure) and is not as detailed as the one created for the interaction model. The execution model contains UML activity diagrams and the interaction model contains UML collaboration diagrams.

Decision Model

The decision model contains information about how the models change for the different applications. The structural model describes the classes and relationships by which a Komponent interacts with its environment, as well as any internal structure of the Komponent, which is visible at its interface.

(The most significant difference between KobrA's decision models and those of other product line approaches is their integration into the composition hierarchy of a framework.)

Structural Model

The structural model is composed of UML class diagrams and UML object diagrams. Class diagrams define the classes, attributes, and relationships that describe the externally visible types characterizing the component's relationship to its environment. Object diagrams are only needed if the component under specification contains white box components. If this is the case, the purpose of the object diagrams is to describe the parts of the internal structure that are externally visible.

Behavioral Model

The behavioral model describes how a component reacts in response to external stimuli. It consists of an arbitrary number of UML state chart diagrams and an optional event map.

Functional Model

The functional model of a Komponent describes the externally visible effects of the operations that are provided by that component. It consists of a set of operation schemata. Each operation listed in the class diagram must have a corresponding operation schema which defines its effects in terms of input parameters, changed variables, output values (reads, changes, and sends clauses), as well as pre- and post conditions (assumes and result clauses).

Containment Trees

The containment relationship, defines the nesting of Komponents within each other based upon their module-like properties. The other, clientship, defines the "uses" dependencies between Komponents that arises when instances of one invoke the services of instances of another. Komponent trees should be understood in relation to UML package structures. Because of the principle of locality (see above for all 4 principles), in all but the simplest containment trees no individual Komponent "knows" about the shape of the whole tree. An individual Komponent only knows who its ancestors and its immediate children are. Because containment relates to the module-like properties of Komponents, the UML depiction of a containment tree uses packages to represent Komponents.

2.1.3.3 Activities

KobrA's product line engineering process consists of two primary sub-processes: the framework engineering process and the application engineering process. They are further decomposed into a set of activities and sub-activities. The framework engineering process consists of the 8 following activities:

- Context realization
- Komponent specification
- Komponent realization
- Komponent implementation
- Component reuse
- Inspection
- Measurement of structural properties
- Testing

Application engineering consists of 10 activities:

- Application context realization
- Application Komponent specification
- Application Komponent realization
- Application Komponent implementation
- Construction
- Component building
- Releasing
- Inspecting
- Measurement of structural properties
- Testing

See [Atkinson02] for a complete description of all artifacts involved.

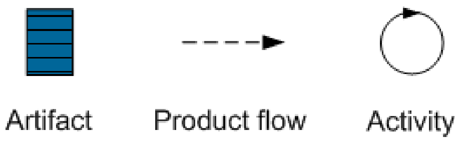


Figure 2-4: Notation in figures

The product flow between the framework engineering and application engineering is illustrated in figure 2-5 (see figure 2-4 for explanation on the notation).

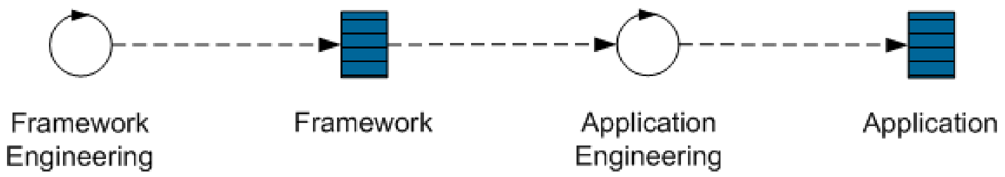


Figure 2-5: Product line engineering in KobrA

It is important to note that figure 2-5 does not imply sequential execution. The process model only defines the static structure of the process, not its dynamic performance. The figure does however show that application engineering needs input from the framework, but there is nothing that stops the framework engineering to continue in parallel with application engineering or be re-entered later to make changes to the framework. Figure 2-6 (see figure 2-4 for explanation on the notation) shows the framework engineering in KobrA. All 8 activities are drawn together with the artifacts involved. Each of the activities can be decomposed with even more sub-activities. See [Atkinson02] for an overview of the application engineering activities in KobrA.

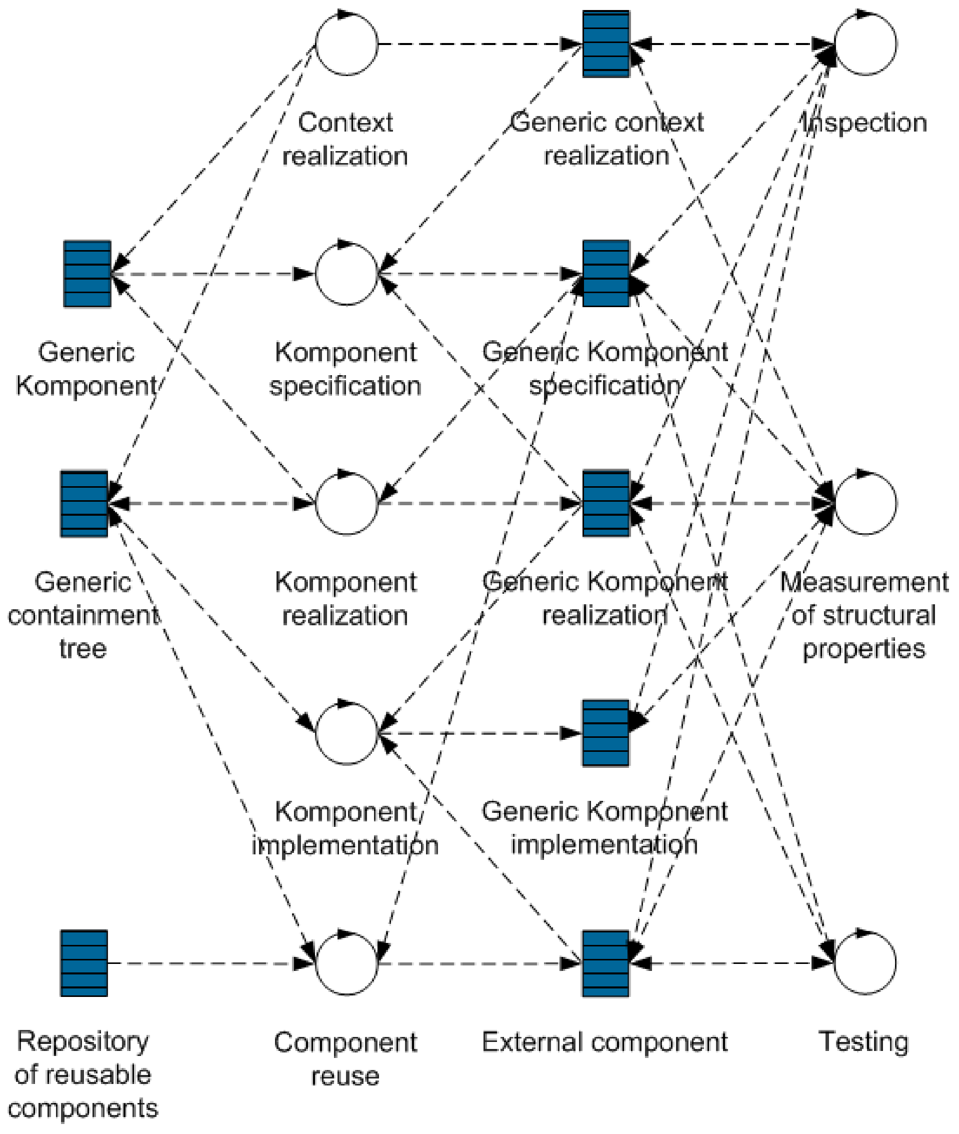

Context realization
Generic context realization
Inspection
Generic Komponent
Komponent specification
Generic Komponent specification
Generic containment tree
Komponent realization
Generic Komponent realization
Measurement of structural properties
Komponent implementation
Generic Komponent implementation
Repository of reusable components
Component reuse
External component
Testing

Figure 2-6: Framework engineering in KobrA

2.1.3.4 Component Identification

The Komponent identification activity is performed at the end of the realization activity in order to identify potential new sub-components. This activity is needed because the primary realization activities do not always discriminate between classes or Komponents, the choice between a simple class and Komponent is made automatically. Typically simple classes have less size and many instances. However, new classes, introduced into the system for the first time as part of the realization in question, can either be treated as simple classes or Komponents. It is the job of the realization activity to make the choice.

Komponent identification is a sub activity of the realization activity (called Komponent realization in figure 2-6 on page 22), because it must be performed before the final version of the structural model can be created with the appropriate application of stereotype <<Komponent>> where necessary. However, it takes place only after the other activities, except quality assurance, have been completed.

Variant functionality

The KobrA method explores different ways of organizing and structuring parts of a software system. Variant functionality cannot always be so easily encapsulated within a single Komponent. To control variability, [Atkinson02] suggests that variant model elements and corresponding decisions have to be included in most of the Komponents in the framework. Encapsulation of variability is not the only criterion for identifying Komponents within a framework containment tree, but it is an important one, as described in [Atkinson02]: *"In general, there is no easy cookbook for identifying good Komponents in a project, and much depends on the experience and domain knowledge of the engineers. Analyzing commonalities and variabilities is an important tool in the arsenal of Komponent engineers, especially when it comes to the identification of highly reusable components".* Again (as in GSN RUP) we can see the importance of having experienced engineers to effectively identify reusable components, or in this case Komponents. The same applies when creating reusable components, although KobrA does lay out a few guidelines like exploring variant functionality described above.

Variabilities are characteristics that may vary from application to application. In general, all variabilities can be described in terms of alternatives. At a coarse-grained level, one artifact can be seen as an alternative to another artifact. Then during application engineering, the artifact that best matches the context of the system under development is selected. Although simple in theory, providing an effective representation of the variabilities in a product family is one of the most critical, and also problematic, factors in the success of a product line engineering project. The analyzing and encapsulating of the variabilities that characterize a product line can provide valuable insight into how best to consolidate functionality into good, reusable building blocks for the domain in question. In short, factoring out and encapsulating variabilities can be a good way of identifying components.

The first issue that must be resolved when analyzing variability is whether it is to be realized as a run time variability or development time variability. By their very nature, software systems are inherently full of variabilities because their execution paths are driven largely by run time input. Product line engineering, in contrast, is concerned with development time variabilities that are resolved before a system is loaded onto its final execution environment. The problem is that it is no strict boundary between development time and run time variabilities. It could be possible to realize all variabilities at run time, that is to build all features of the entire domain into a single system with appropriate parameters to select the desired features at run time. However this would not be regarded as product line engineering. On the contrary, one of the goals of product line engineering is to avoid code bloating which results from an attempt to resolve all variabilities at run time.

Variability identification is done in all the first 4 activities of the framework engineering activity (see chapter 2.1.3.3 above for a full overview of the activities).

Identification of reusable components

KobrA component specification (see chapter 2.1.3.2) provides comprehensive but abstract descriptions not only of what a component provides but also what it expects. This is important for being able to correctly identify reusable components. [Atkinson02] lists four main categories of assumption that can cause architectural mismatches. They are:

1. nature of components
2. nature of their interactions
3. nature of the global architectural structure
4. construction process

The KobrA component specification model attempts to address all of these. Although the authors of [Atkinson02] states: *"KobrA does not address the problem of finding components (although KobrA specifications could be helpful in this regard), but only focuses on introducing components into a containment tree once they have been found."*

The final goal of the component reuse activities in KobrA is to fully integrate a component developed earlier outside the tree into the Komponent tree. To achieve this, the specification desired by the reusing Komponent and the specification offered by the pre-existing external component have to be brought into agreement. This is true for both Komponents and COTS developed without KobrA.

The process of identifying reusable components takes place in the activity Component reuse (see figure 2-6 on page 22). This activity consists of 4 sub-activities:

1. Component selection
2. Negotiation
3. Containment tree adaptation
4. Component adaptation

as well as artifacts as Repository of reusable components. See [Atkinson02] for a complete specification on this activity.

2.1.3.5 Summary

KobrA is an object oriented analysis and design methodology for all phases of software development. KobrA provides a methodology for the full life cycle of software development, and is in our opinion designed to fit well with RUP. Although we see that KobrA's product line approach is somewhat different from the RUP, the managerial activities that accompany the technical product line engineering activities, such as project management, configuration and change management and quality assurance are quite similar. KobrA is much less prescriptive about the management aspects of a software project than it is about the technical aspects. In this sense KobrA is compatible with RUP and the GSN RUP adaptation.

The variant functionality exploring described and used in KobrA (as well as in other methods) can help to identify components. The analyzing and encapsulating of the variabilities that characterize a product line can provide valuable insight into how best to consolidate functionality into good, reusable building blocks for the domain in question. Factoring out and encapsulating variabilities is in fact a way of identifying components. This could help in the process of evaluating what components that should be added to the framework. (See the text on variant functionality on page 23 for a more in depth description.)

The component modeling aspects of KobrA is designed to provide a more rigorous and component oriented approach for using the current UML standard.

Advantages:

- KobrA provides a component definition model that fits well and fulfills the existing UML-model, alleviating some of the most problematic issues with pure UML design.
- KobrA is focusing on the technical aspect of software development and is therefore more or less easily integrable with other processes with a focus on managerial aspects i.e. RUP.
- KobrA provides a full life cycle methology, making the need for additional processes for a complete methology obsolete.
- KobrA provides a separate product line process giving specific guidelines on how to reuse components.

Disadvantages:

- Finding sources that have experience in using KobrA is difficult. The method is fairly new and has probably only been adopted by industry partners. Users with hands on experience are necessary to complete the image whether or not this process is better or worse than i.e. RUP.
- KobrA does not fully deal with all the issues of component identification. It provides a method that could help in this process, but still (as most other methods) it relies on the experience of the designers/engineers.

2.1.4 COMET

The information in this chapter is gathered from [Solberg97].

Component-based Methodology (COMET) is developed as part of the Open Business Object Environment (OBOE) project submitted under the ESPRIT framework IV. One of the objectives of the OBOE project is to implement the OMG's envisioned Business Object Facility (BOF). BOF is an advanced platform for Business Objects. The OBOE team includes Schlumberger Geco-Prakla (User), SSA Object Technology (Technology supplier), Prism technology (Technology supplier), Heinz Nixdorf Institute (User group), University of Frankfurt (User group), SINTEF (Methodology, User group), Log-On Technology (Marketing) and X/Open (Coordinator). The partners' roles in the project are indicated in the parenthesis.

COMET is characterized by development of application-independent business object models, and the realization of these models in software components. It is a stepwise description that gives guidelines of how to analyze, design and implement software systems. It has a focus on how to do this in distributed systems. For the realization, COMET recommends the OMG's Business Object Facility (BOF) is used. An implementation of this platform, on top of CORBA, is also under development within the OBOE project. A large part of the methodology, however, is independent of the implementation platform.

COMET consists of three main modeling phases: business modeling, component-modeling and implementation modeling. These phases are traversed more or less in this order, although the method does not prescribe a strictly linear process: like most methods, it allows incremental and iterative development of software. Each of the models consists of a number of sub models, as shown in figure 2-7 below.

In each of the three phases, a reuse model is used to explicitly address the reuse of components. This model is orthogonal to the other models, and uses the notion of a reuse repository. This repository is used in two ways:

- Search the reuse repository for reusable components, frameworks, design patterns etc. that may be used.
- Identify possibly reusable components, frameworks, design patterns etc., and once they have been developed, add them to the reuse repository.

COMET is characterized by development of application-independent business object models, and the realization of these models in software components. COMET differ with traditional analysis and design methods in that the method create and maintain a linkage from business and business strategy to software, so the software components reflect the business and the business strategy. One of the driving values behind COMET is that information systems can and should be assembled from components.

COMET is a model driven methodology. There are four main models in COMET: The Business Model, the Component Model, the Implementation Model and the Reuse Model. These models include sub models to separate finer grained concerns. The sub models include certain activities to create certain deliverables. These are called work products. The COMET models and their overall relations are depicted in figure 2-7 below.

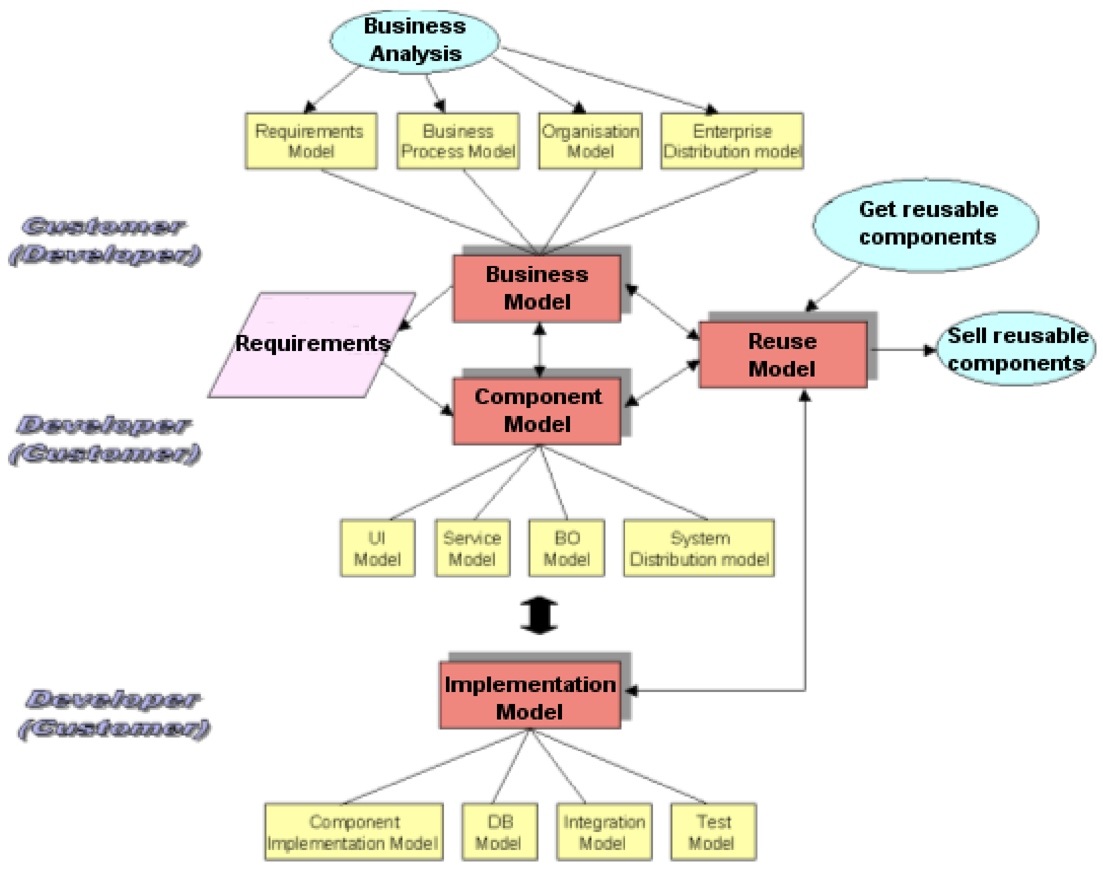


Figure 2-7: COMET models and their overall relations

Business Model

The model includes business analysis and modeling and identification of goals and requirements. The business model is separated in four sub models:

1. The Requirements Model concerns identification of the Area of Concern and identification requirements (requirement engineering). The requirements may be grouped into functional requirements, non-functional requirements and external constraints. Use Case modeling is used to capture the functional requirements and is a main activity in the Requirements Model.
2. The Business Process Model concerns identification and analysis of activities and business processes. Business process modeling and scenario modeling is main activities in the Business Process Model.
3. The Organisation Model concerns the organization structure and identification of roles and their responsibilities.
4. The Enterprise Distribution Model concerns the distribution of the Enterprise, for instance distribution of installations, workspaces, general resources, roles and so on.

The business analysis binds the four sub models of the Business Model together. All four models include some business analysis activities.

Component Model

The model includes design of user interface, services, business objects and other components. Prototyping is also part of the Component Model. The Component Model is separated in four sub models:

1. The UI model concerns design of user interfaces and prototyping.

2. The Service model concerns description and design of services.
3. The Business Object Model concerns identifying, classifying and designing the Business Objects.
4. The System Distribution Model concerns distribution of the components.

Implementation Model

The model includes implementing and integrating and testing the system. The Implementation Model is separated in four sub models:

1. The Component Implementation Model concerns implementing the components from the component model.
2. The DB Model concerns determining the persistent object model, database mapping and implementation of the database structure.
3. The Integration Model concerns the integration of the system.
4. The Test Model concerns testing the system and its sub systems and components.

Reuse Model

The model includes identifying and producing reusable components, patterns frameworks, analysis models, design models etc, as well as offering these in a reuse repository. COMET encourages development *with* and *for* reuse. The Reuse Model is orthogonal to the other models. Actors working in the other models are responsible for using reusable assets whenever possible. This is done by searching the reuse repository. They are also responsible for suggesting new reusable constituents to the Reuse modelers.

Since COMET encourage iterative development, the Business Model, the Component model and the Implementation Model will give input to each other. Sub models developed in one model will typically serve as basis for sub models in the other model.

As shown in figure 2-7 an important output of the Business Model is a set of requirements. This is typically functional requirements, non-functional requirements and external constraints. Example of external constraints is budget, available resources, time constraints, strategy decisions, customer policy etc. The requirements will be main input to the Component model.

For the Business Model the Customer will play the main role, while the developer will play the main role in the Component model. Both the customer and the developer will participate in the development of both these models. The responsibility for the Implementation Model belongs mainly to the developer, but the customer should be involved in the Test Model.

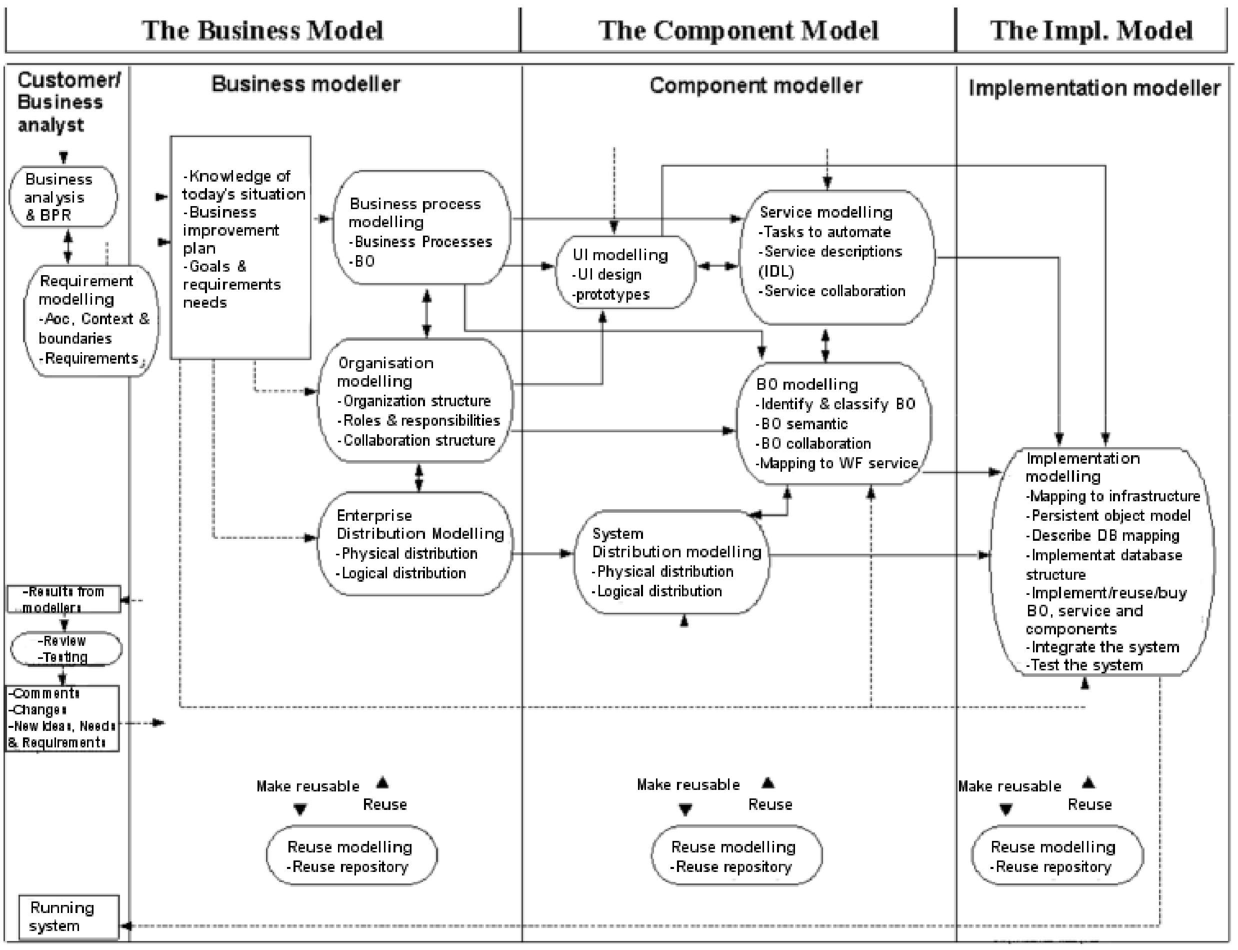

The Business Model
The Component Model
The Impl. Model
Customer/ Business analyst
Business modeller
Component modeller
Implementation modeller
Business analysis & BPR
Requirement modelling -Aoc, Context & boundaries -Requirements
-Knowledge of today's situation -Business improvement plan -Goals & requirements needs
Business process modelling -Business Processes -BO
Organisation modelling -Organization structure -Roles & responsibilities -Collaboration structure
Enterprise Distribution Modelling -Physical distribution -Logical distribution
UI modelling -UI design -prototypes
Service modelling -Tasks to automate -Service descriptions (IDL) -Service collaboration
BO modelling -Identify & classify BO -BO semantic -BO collaboration -Mapping to WF service
System Distribution modelling -Physical distribution -Logical distribution
Implementation modelling -Mapping to infrastructure -Persistent object model -Describe DB mapping -Implementat database structure -Implement/reuse/buy BO, service and components -Integrate the system -Test the system
-Results from modellers
-Review -Testing
-Comments -Changes -New ideas, Needs & Requirements
Make reusable
Reuse
Reuse modelling -Reuse repository
Running system

Figure 2-8: COMET process overview modeled with UML activity diagram

The headings above the UML activity diagrams show which part of the activity diagram that belongs to the Business Model, the Component Model and The Implementation Model. The Reuse Model is orthogonal to these models and interacts with all the processes in the activity diagram.

2.1.4.1 Component Definition

The process of component definition uses the descriptions of the business objects as the starting point and describes the business object semantics. The Business Object (BO) semantic may include concepts like:

- Roles
- Events
- States
- Properties
- Rules
- Relations
- Adapters
- Dependency
- Exceptions

Business Objects are defined as: "*A business object represents a concept or entity in an application domain. Business objects should be known to and visible (in some way) to the users of applications in the domain. Business objects have name, attributes, behavior, relationships, rules, policies and constraints. BO's is able to communicate with each other at a semantic level, and is self-contained deliverables that know how to co-operate with other separately developed business objects to perform a desired task. Business objects also encourage a view of software that transcends tools, applications, databases, and other system concepts.*"

Today ODL (Object Definition Language) is probably a preferred notation may be with some extensions, but if the OMG's proposed CDL (Component Definition Language) matures and becomes a standard this language may be preferable. In addition to this description a collaboration structure must also be described.

Collaboration structures developed in the collaboration structure work product in the Business Process Model must be extended and refined. The most convenient way of describing this is by using business object collaboration, UML Collaboration diagrams, UML activity diagrams and UML scenario diagrams.

The Component Based Methodology Handbook [Solberg97] recommends that development projects consider mapping the process business objects to workflow services implemented in and workflow engine if this is a relevant consideration in the actual project. The notation used for this is free text and links to process business objects.

2.1.4.2 Component Identification

COMET is a methodology for and with reuse. The Reuse Model is orthogonal to the other models in COMET and for each activity in each work product reuse considerations should be taken into account. This has two aspects. One is to investigate if there are any reusable components, frameworks, design patterns and so on that may be used in a specific work product. Search the reuse repository to do this. The other is to identify possible reusable components, frameworks, design patterns and like and eventually develop those and adds it into the reuse repository.

The Reuse Model concerns maintaining the reuse repository, and in this model the final decision of which of the reusable candidates that shall be added into the reuse repository is made. Reuse candidates are made reusable in this model. This means that making reusable components, frameworks, patterns and so on is a dedicated task normally done by dedicated people as part of the Reuse Model. The authors of [Solberg97] express that experience has shown that activities in the reuse model should be separated from the system development project. That is, activities in the reuse models should have its own budget and so on. The main project concerning development of the software system should not be burdened with use of resources to develop general reusable components, frameworks, patterns and so on, although reuse candidates should be identified and sent to the reuse modelers.

Figure 2-9 shows how the COMET models relate to the traditional Analysis, Requirement, Design and Implementation phases and what which are typically reused in those phases.

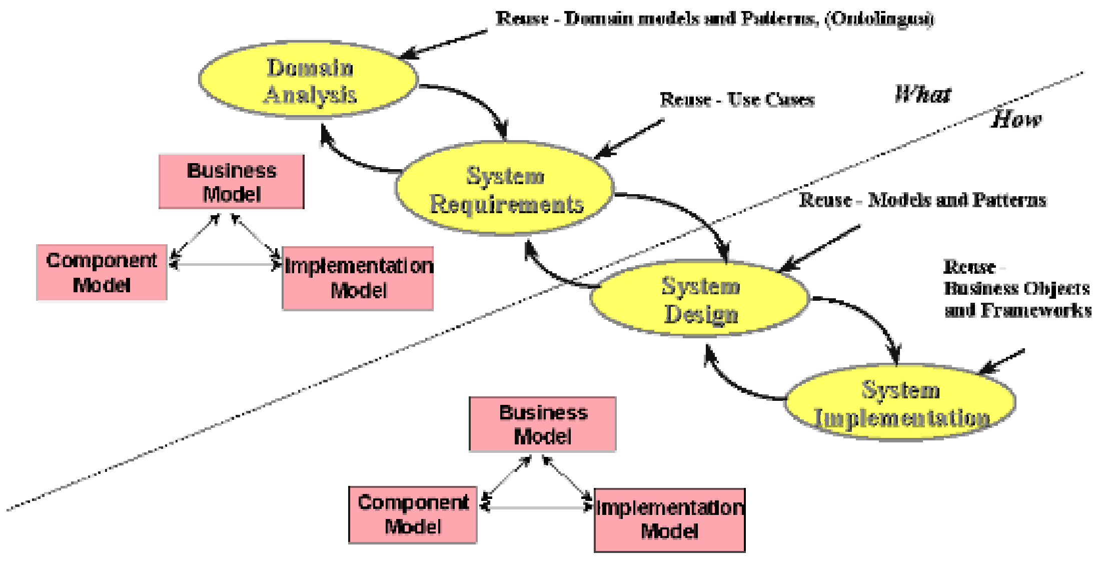


Figure 2-9: How COMET supports evolutionary development and reuse

All four models in COMET will be considered and evolve during the analysis design and implementation phases. Some models will of course have more focus than others will in a specific phase though. For instance the Business Model has more focus than the Implementation Model in the domain analysis phase.

The figure illustrates that the phases are not sequential. The work iterates between the phases. This supports the evolutionary nature of software engineering development. What to reuse in different phases are also depicted in figure 2-9, and is the responsibility of the Reuse Model.

2.1.4.3 Summary

COMET is an object oriented analysis and design methodology that focus on using BO and business concepts during the analysis, design and implementation phases. We feel that the COMET process is too specialized in the direction of being OBOE and OMG compliant rather than providing an open mind as to which processes that can use it. This is because it is prescriptive about both the management aspects and the technical aspects of a software project. COMET is not directly integrable with RUP. In addition COMET does not have a full life cycle methodology. (COMET is essentially dealing with OOA, OOD and OOP, and do not focus on program planning, project planning, evaluation and implementation planning.)

Advantages:

- COMET attempts to provide a method to create and maintain a linkage from business and business strategy to software. This is thought to make the software components reflect the business and the business strategy.
- Provides a description of both managerial and technical aspects.

Disadvantages:

- Does not provide a full life cycle metholody, making the need for additional processes to have a complete metholody.
- Finding sources that have experience in using COMET is difficult. According to [Boertien01] the only place where this metholody has been used is in user case within the OBOE project. Users with hands on experience are necessary to complete the image whether or not this process is better or worse than i.e. RUP.

2.2 Standards

This chapter contains a study of the state-of-the-art of standards for component specification. We begin with looking at the current UML version, followed by a study of the new UML 2.0 standard. This chapter is followed by a description of the CORBA Component Model (CCM) and finally a chapter on the Reusable Asset Specification (RAS).

2.2.1 Unified Modeling Language (UML)

This chapter begins with an introduction to the Unified Modeling Language (UML) and a study of the support for components in the existing version of UML. We then address the need for the revised UML 2.0 specification followed by a description of component specification in the proposed UML 2.0 standard. We have chosen not to present all the other new constructs of UML 2.0 since this would be outside the scope of this report.

2.2.1.1 Introduction to UML

The Unified Modeling Language (UML) has been widely accepted throughout the software industry and successfully applied to diverse domains ever since it was adopted by the Object Management Group (OMG) in 1997. Since then, UML has now become the de facto standard for visualizing, specifying, constructing and documenting software systems.

Diagrams in UML abstract software details and visualizes the structure, communication and interaction between objects. The visualization of software is a way of avoiding the complex nature of software in stages before and after code.

UML can help planning and documenting large and complex software systems. Knowing and using UML will help analytics, designers, architects, developers, customers i.e. communicate about software.

2.2.1.2 UML 1.x Diagram Support

UML 1.x supports the following diagrams:

- **Use Case Diagram:** Captures the system functionality from the standpoint of an external observer. Each use case describes a usage scenario.
- **Class Diagram:** Gives a static overview of system classes and their relationships.
- **Behavior Diagrams:**
    - **Sequence Diagram:** Captures the time-related dynamic behavior of a system. The diagram describes in detail how operations are carried out.
    - **State diagram:** Captures the event-based dynamic behavior of a system and the transition between different states.
    - **Activity Diagram:** Much like a state diagram, but focuses on the flow of activities involved in a single process instead of objects states.
    - **Collaboration Diagram:** Describes the message interaction between objects. It illustrates coordination of elements in response to specific events.

- **Package Diagram:** Captures the decomposition of modeling elements into packages, and dependencies among packages. These packages can be used to increase the understanding of the main elements of the system when the systems get complex.

- **Implementation Diagrams:**
  - **Component Diagram:** Captures the physical structure of the implementation of a system. It is built as a part of the architecture of a system to organize and manage the physical elements of a solution. A component is represented as a rectangle with two small rectangles on the left side.
  - **Deployment Diagram:** Captures the physical layout of hardware and software in a system.

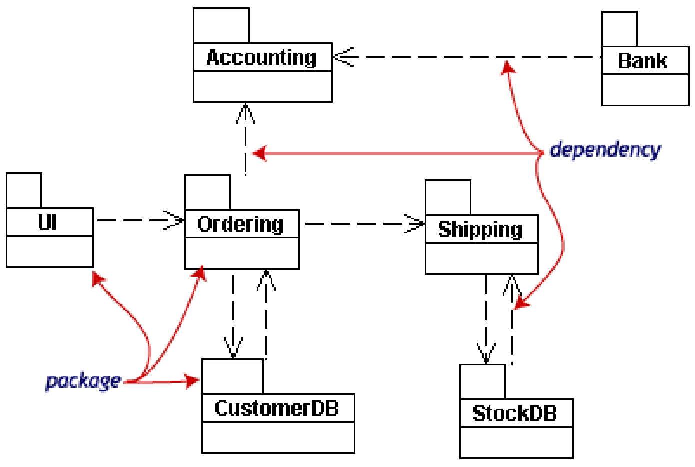


Figure 2-10: Package Diagram example

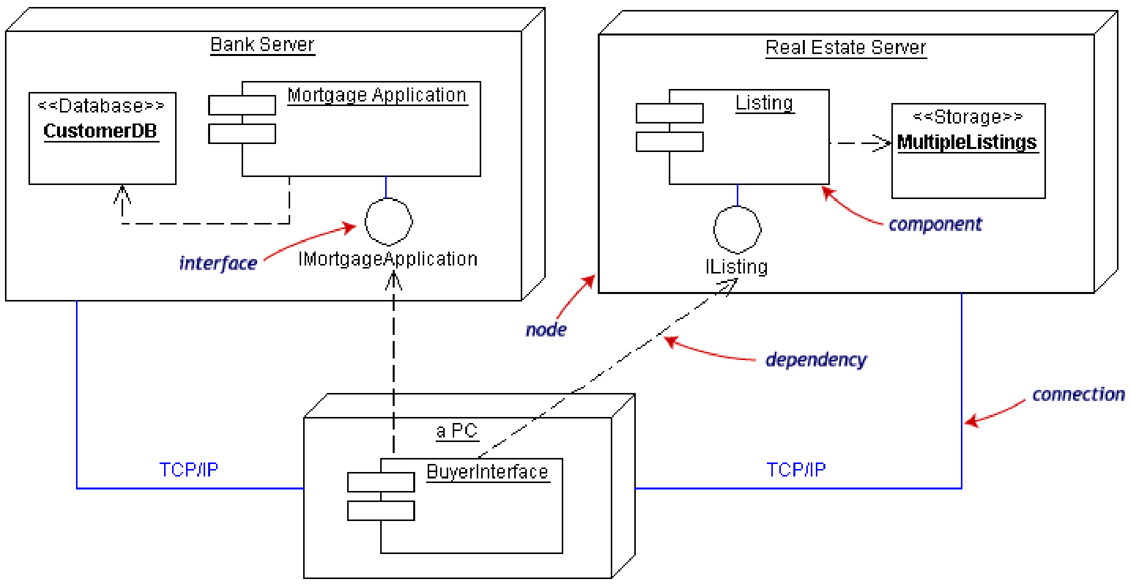


Figure 2-11: Deployment Diagram containing Component Diagrams

Refer to [Fowler00] for more information about UML.

2.2.1.3 Support for Component Specification in UML 1.x

[Cheesman01] UML was originally designed as a language for object-oriented analysis and design, and it is based on the assumptions of an object-oriented implementation. In the current version of UML, the concept of component is geared towards implementation and deployment. But our focus in this study is on component specification, and the current version of UML does not support this construct.

In order to work around this shortcoming, a component is defined as a *stereotype* of a class. Stereotypes are ways of letting designers create new kinds of UML modeling elements. [Cheesman01] defines a component specification as a stereotype of a class called <<comp spec>>, and a UML component realizes this <<comp spec>> class. A component specification offers a set of interface types. At the implementation level this "offers" relationship is modeled as a realization of a set of interfaces by a component (see figure 2-12).

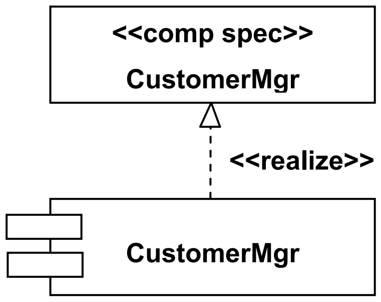


Figure 2-12: Component realizing a component specification

But "realization" is not the correct semantics here, so [Cheesman01] defines a new stereotype of UML dependency called "offers" to capture this important relationship (using a circle notation), see figure 2-13.

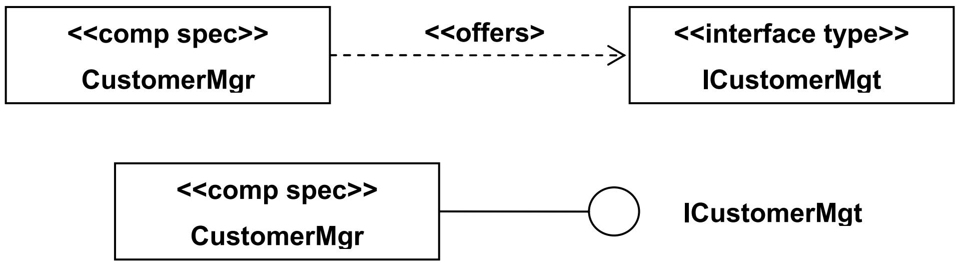


Figure 2-13: Component offering interface types

In addition to a set of interfaces a component offers, it defines a set of interfaces it must use. This is an architectural constraint, not a component design choice. [Cheesman01] models this as a simple UML usage dependency. A usage dependency between a component specification and an interface type specifies that all realizations of the component must use that interface (see figure 2-14). Figure 2-14 is called a component specification diagram. It focuses on a single component specification and details its individual dependencies.

A component specification also defines any constraints that it requires between interfaces, and any constraints on the implementation of its operations. These constraints can be defined using component interactions, and is modeled using UML collaboration diagrams. These diagrams focus on one particular component object and show how it uses the interfaces of other components, or they can show a complete set of interacting components in a complete architecture.

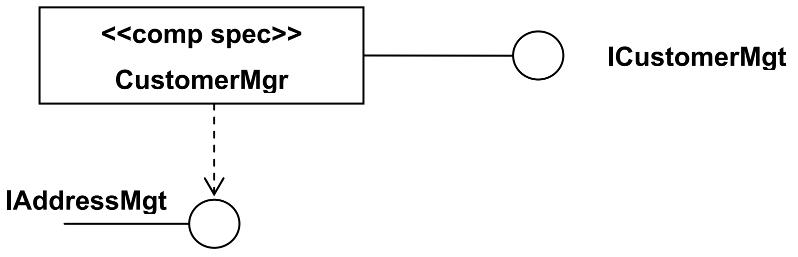


Figure 2-14: Component specification diagram

In our opinion, the workaround using stereotypes to specify components is a shortcoming of the current 1.x version of UML. The lack of support for component specification is also one of the main reasons why the current version of UML now is being revised.

2.2.1.4 Why a major revision of UML was needed

While the UML has been growing in popularity among software developers, the trends within the software engineering field have also been evolving steadily. Especially the transition from Object Oriented Software Engineering to Component-Based Software Engineering (see 2.1.1) is significant.

Some problems with UML 1.x cited in [Kobryn02] include excessive size, high complexity, imprecise semantics, non-standard implementations, limited customizability, inadequate support for component-based development, and inability to interchange model diagrams.

Considering these trends, as well as numerous requests for UML improvements, the OMG issued a Request for Information (RFI) for UML 2.0 in June 1999. By the submission deadline in December 1999, 26 vendors, users, academics, consultants and other standards organizations had submitted their improvement suggestions on UML, and the respondents suggested that a major revision was urgent. [Kobryn00] lists the most prominent requests:

- Support for component-based development
- Precise and unambiguous language kernel
- Additional concepts layered on top of kernel
- Compliance with the Meta Object Facility (MOF)
- First-class extensibility mechanism
- Internal structure of classifiers/interfaces/classes/types/roles
- Limit associations to context
- Meta model for the Object Constraint Language (OCL) to integrate better with UML meta model
- State machine generalization
- Scalability and encapsulation of state machines
- Scalability, encapsulation and structuring of collaboration and sequence diagrams
- Modeling of architectures
- Abstract data flow modeling

- Rigorous mapping from notation to abstract syntax

In September 2000 the OMG issued four RFPs (Request for Proposal) for UML 2.0: an Infrastructure RFP concerned with restructuring the basic constructs and improving customizability; a Superstructure RFP [OMG02-1] to improve more advanced constructs such as components, activities, and interactions; an Object Constraint Language RFP concerned with increasing the precision and expressive power of UML's constraint language; and a Diagram Interchange RFP to address making model diagrams interchangeable between tools. The Superstructure RFP [OMG02-1] is of most interest to our study since it contains the new component specification constructs.

A consortium of companies that are members of the Object Management Group (OMG), called the U2 Partners, submitted a joint RFP to OMG in August 2001. The submission team consists of companies that played key roles in the definition of the original UML standard and its subsequent evolution. On September 9th 2002 the U2 Partners submitted revisions of their UML 2.0 Superstructure and Infrastructure proposals to the OMG. This is the latest news update on the work on UML 2.0.

Since the new UML standard is not yet finished, our study of UML 2.0 is based on the latest RFPs that have been submitted to the OMG. We found little information on UML 2.0 apart from the official OMG information and the revised RFPs. Only a few articles describe the need for UML 2.0 and the features that most likely will be incorporated in the new version.

2.2.1.5 Component Specification in UML 2.0

The new component specification in UML 2.0 is described in chapter 16 of the superstructure RFP [OMG02-1], and information and figures in this chapter is acquired from this document.

Component Definition

According to [OMG02-1], a component is considered as an autonomous unit within a system or subsystem. It has one or more ports, and its internals are hidden and inaccessible other than as provided by its interfaces. Although it may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible. As a result, components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together via their provided and required interfaces. The aspects of autonomy and reuse also extend to components at deployment time. The artifacts that implement component are intended to be capable of being deployed and re-deployed independently, for instance to update an existing system.

The Components package supports the specification of both logical components (i.e. business components, process components) and physical components (i.e. EJB components, CORBA components, COM+ and .NET components), along with the artifacts that implement them and the nodes on which they are deployed and executed.

Semantics

A component is a self contained unit that encapsulates the state and behavior of a set of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components in the system. The required and provided services are defined through ports that are characterized by interfaces. A component does not realize an interface directly, instead, one its parts jointly realize all interfaces offered by the component. Thus, a component may be said to implicitly realize the provided interfaces associated with its ports. The invocation of the behavioral features offered at a port by a provided interface must only be requested through that port. There must be a connector to a part the type of which realizes an interface offering this service. An encapsulated classifier uses the required interfaces associated with its ports either implicitly or explicitly.

A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its port definitions. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality.

A component has an *external view* (or "black-box" view) by means of its publicly visible properties which are defined through ports. A port represents a named interface that the component either provides to its environment or requires from its environment. A component also has an *internal view* (or "white-box" view) by means of its private properties which are its internal classifiers and how they are connected. This view shows how the external behavior is realized internally. The mapping between external and internal view is by means of delegation connectors behind the ports that are connected to internal parts (i.e. sub-components or classes).

Notation

A component is shown as a classifier rectangle with the keyword «component». Optionally, in the right hand corner a component icon can be displayed.

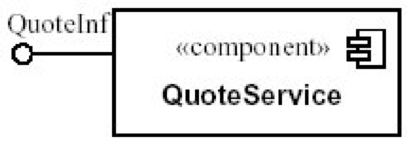


Figure 2-15: A Component with one provided interface.

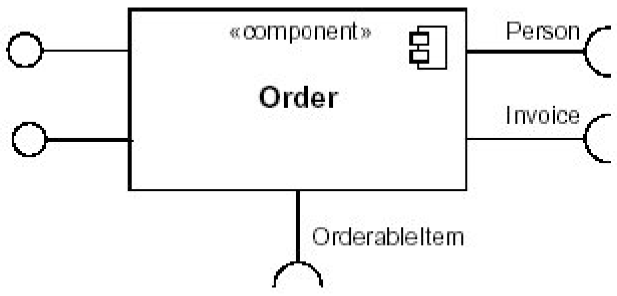


Figure 2-16: A Component with two provided and three required interfaces.

An external view of a component is by means of Interface symbols sticking out of the component box (external or black-box view). Alternatively, the interfaces can be listed in the second compartment of a component box. This notation can be used for overview diagrams where components expose and require large sets of interfaces.

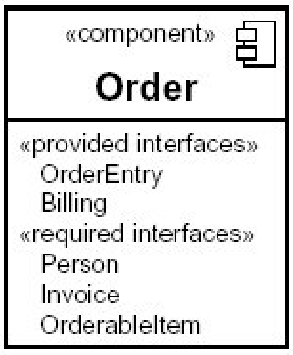


Figure 2-17: Alternative black box notation of a component.

The internal structure of a component can be shown in more detail by viewing its parts and connectors. Classifiers that are owned or imported by a component can be shown nested inside the component. Optionally, explicit part names or connector names can be shown in situations where the same classifier or association is the type of more than one part (connector).

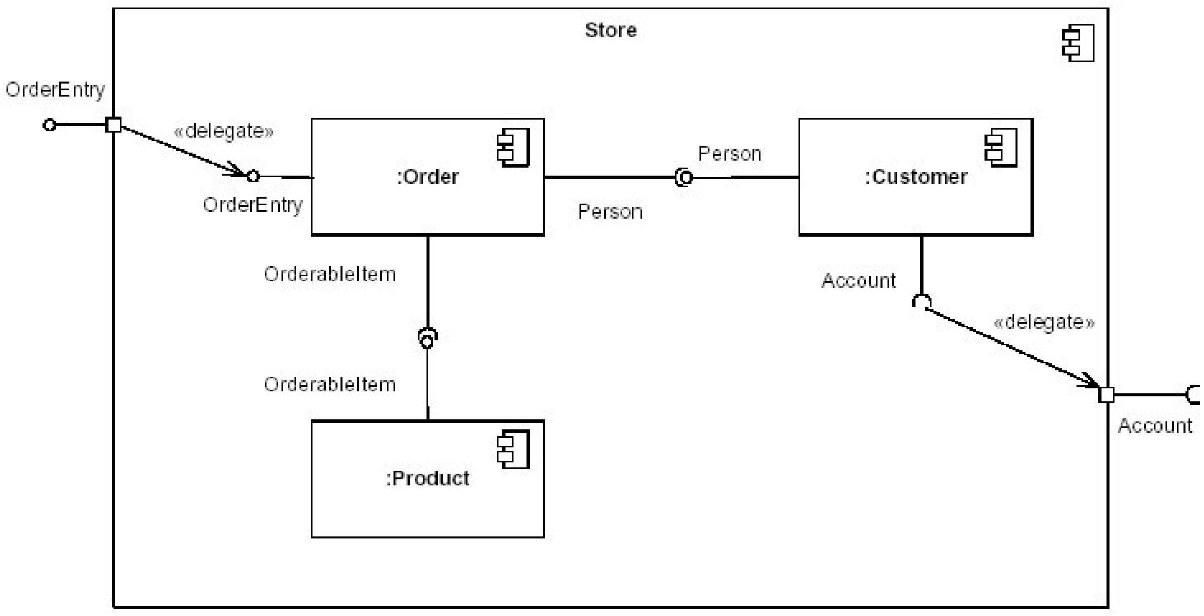


Figure 2-18: Component diagram showing a white-box view of a 'nested' component

Component Relationships

A component specification is a static structure diagram with a graph of components that are linked by connector and dependency relationships. Connector relationships are shown by dashed, directed arrows between provided and required interfaces. Optionally, only provided Interfaces can be shown explicitly (in which case the connector relationship originates directly from the requiring component). Dependencies are shown as directed dashed lines from a client component to a supplier component that it depends on in some way.

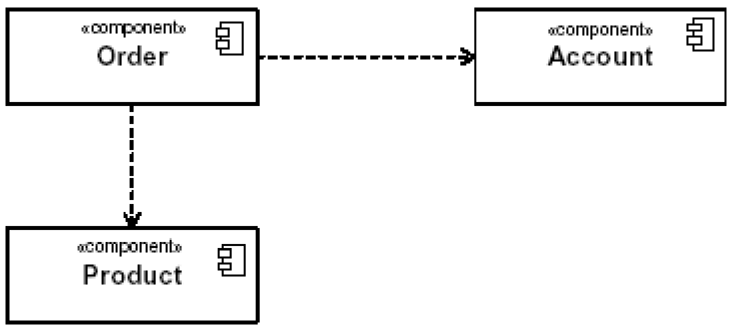


Figure 2-19: Components and their general dependencies

Component Deployment Diagrams

Component deployment diagrams extend the basic deployment diagrams by adding the deployment specification aspects specific to deploying components in execution environments. Component Deployment diagrams are new to UML 2.0.

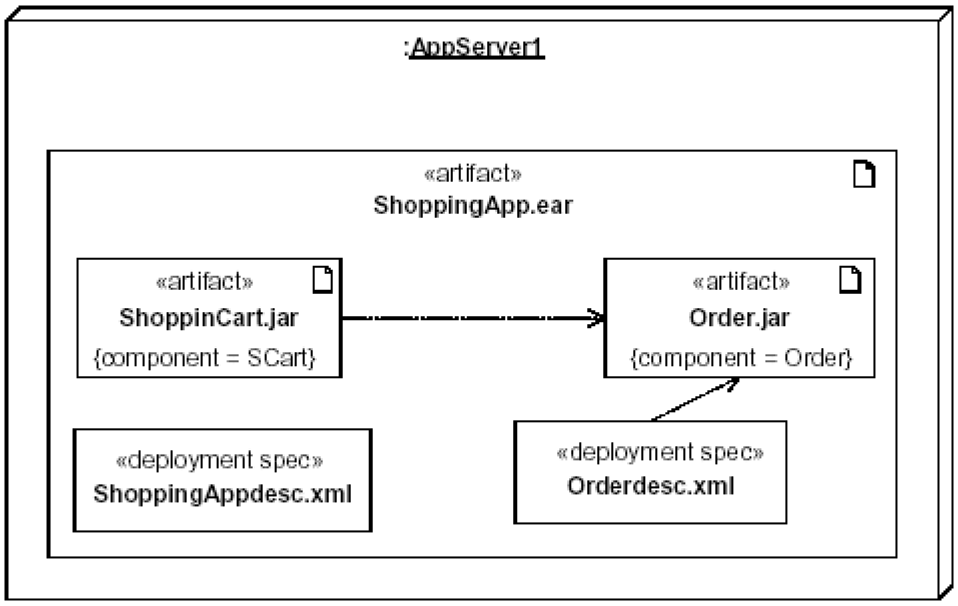


Figure 2-20: Example of a component deployment diagram

2.2.1.6 Sequence Diagrams in UML 2.0

A major problem with sequence diagrams in UML 1.x is that they tend to become very large and complex. In UML 2.0 this issue is addressed by the use of references. References allow interactions to be referenced within other interactions. The use of references hides details in the overview diagram, allowing decomposition of sequences into more detailed ones. Hopefully this will improve the overview and readability. See figure 2-21 for an example of a sequence diagram with references.

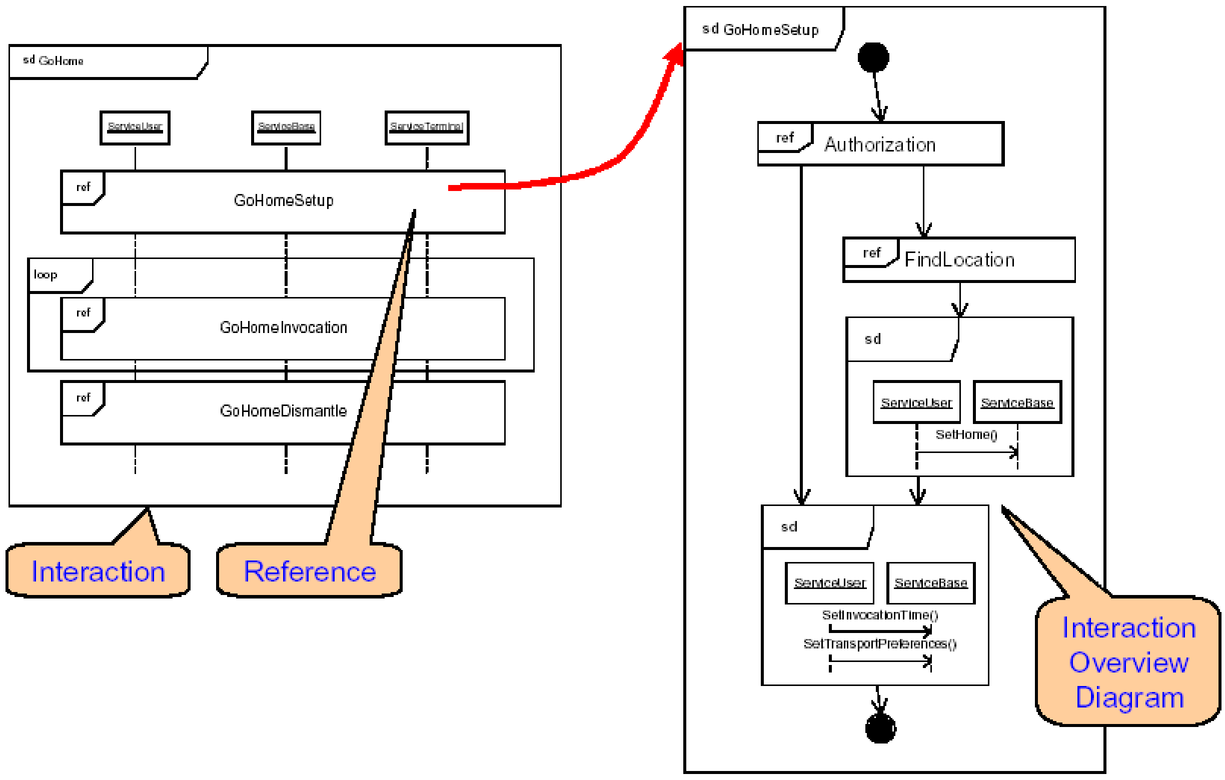


Figure 2-21: Sequence Diagram with reference

2.2.1.7 Summary

In this chapter we have presented some of the new elements of the forthcoming UML 2.0 standard, especially the support for Component-Based Development and the specification of components in the logical view.

We think that the new component specification capability of UML 2.0 is an important improvement of UML. The specification of components earlier in the design phase can now be done in a standardized way without the use of stereotypes and other workarounds. The use of ports (provided and required interfaces) offers flexible reuse and replacement of components. The sequence diagrams of UML 2.0 have also been improved. We think that the practice of Component-Based Development will increase even further with the release of UML 2.0.

Advantages with UML 2.0 versus UML 1.x:

- Supports Component-Based Development.
- Specification of components earlier in the design phase.
- Flexible reuse and replacement of components with the use of ports (provided and required interfaces) allowing “wiring” of components.
- Support for both logical components (i.e. business components) and physical components (i.e. EJB components, CORBA components).
- Improved sequence diagrams and new component deployment diagrams.

Disadvantages:

- Currently not a finished standard and hence no users or tools exist at this time. Further evaluation of UML 2.0 will rely on experiences of future users.

2.2.2 CORBA Component Model (CCM)

This chapter contains an overview of the new CORBA Component Model, and contains information from [OMG02-2] and [Wang00].

2.2.2.1 Introduction to CORBA

Since 1989, the Object Management Group (OMG) has been standardizing an open middleware specification to support distributed applications. The traditional OMG Common Object Request Broker Architecture (CORBA) shown in figure 2-22 enables software applications to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols, interconnections, or hardware [Wang00].

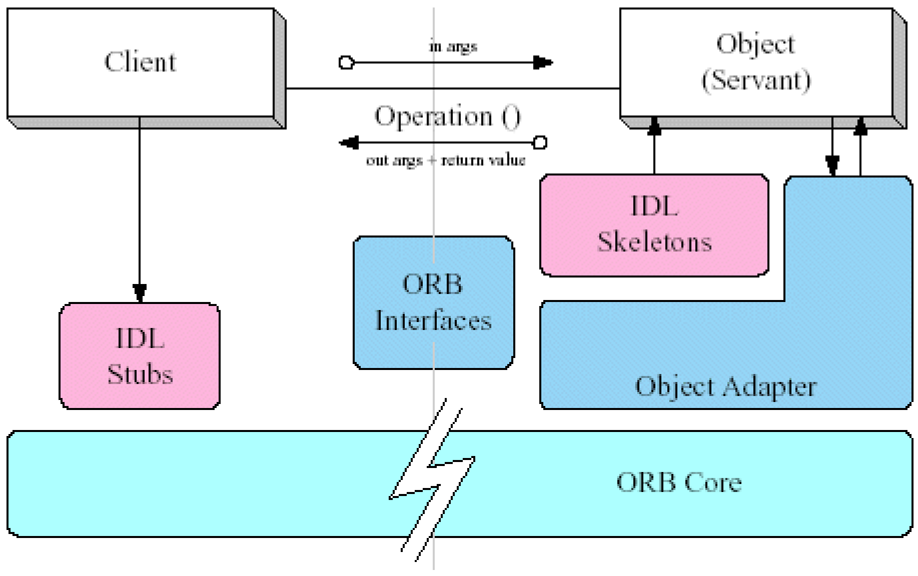


Figure 2-22: Traditional CORBA Object Model

To provide higher-level reusable components, the OMG also specifies a set of CORBA Object Services that define standard interfaces to access common distribution services, such as naming, trading, and event notification. By using CORBA and its Object Services, system developers can integrate and assemble large, complex distributed applications and systems using features and services from different providers. Refer to [Wang00] for a list of limitations with the earlier CORBA object model.

2.2.2.2 Overview of CCM

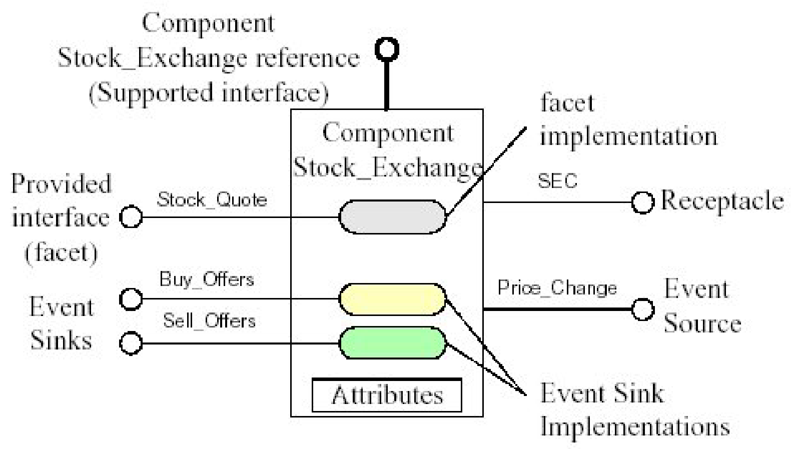
CCM extends the CORBA IDL to support components. The CCM extends the CORBA object model by defining features and services that enable application developers to implement, manage, configure and deploy components that integrate commonly used CORBA services, such as transaction, security, persistent state, and event notification services, in a standard environment. In addition, the CCM standard allows greater software reuse for servers and provides greater flexibility for dynamic configuration of CORBA applications. With the increasing acceptance of CORBA in a wide range of application domains, CCM is well positioned for use in scalable, mission-critical client/server applications [Wang00]. The current version 3.0 of CCM was released in June 2002, so it is a fairly new standard.

2.2.2.3 Component Definition

According to [OMG02-2], a component in CCM is a specific, named collection of features that can be described by an IDL component definition or a corresponding structure in an interface repository. Although the current CCM specification does not attempt to provide mechanisms to support formal semantic descriptions associated with component definitions, they are designed to be associated with a single well-defined set of behaviors. Although there may be several realizations of the component type for different run-time environments (i.e., OS/hardware platforms, languages, etc.), they should all behave consistently. A component type encapsulates its internal representation and implementation. Although the component specification includes standard frameworks for component implementation, these frameworks, are completely hidden from clients of the component.

2.2.2.4 Component Specification

[Wang00] A major contribution of CCM derives from standardizing the component development cycle using CORBA as its middleware infrastructure. Component developers using CCM define the IDL interfaces that component implementations will support. Next, they implement components using tools supplied by CCM providers. The resulting component implementations can then be packaged into an assembly file, such as a shared library, a JAR file, or a DLL, and linked dynamically. Finally, a deployment mechanism supplied by a CCM provider is used to deploy the component in a *component server* that hosts component implementations by loading their assembly files. Thus, components execute in component servers and are available to process client requests. Figure 2-23 shows an example CCM component implementing a stock exchange and its corresponding IDL definition.



```
// IDL code for a CCM Component
interface Sell, Buy; // Define an equivalent, supported interfaces
component Stock_Exchange supports Sell, Buy
{
      provides Stock_Quote;         // Facet
      consumes Buy_Offers;          // Event Sink
      consumes Sell_Offers;         // Event Sink
      publishes Price_Change;       // Event Source
      uses SEC;                     // Receptacle
      ...                           // Other definitions
};
```

Figure 2-23: Example of a CCM component with corresponding IDL code

An object reference hides the location where the actual object resides and contains protocol information defined by the CORBA specification, as well as an opaque, vender-specific *object key* used to identify a servant that implements the object. To developer end-users, the format of a reference to a `Stock_Exchange` component is identical to the format of a reference to a `Stock_Exchange` interface. Thus, existing component-unaware software can invoke operations via an object reference to a component's *equivalent interface*, which is the interface that identifies the component instance uniquely. As with a regular CORBA object, a component's equivalent interface can inherit from other interfaces. This is called the component's *supported interfaces*. In this example, the supported interfaces perform transactions to buy and sell stock. According to [Wang00], it is inflexible to extend CORBA objects solely using inheritance. Thus, CCM components provide four types of mechanisms called *ports* to interact with other CORBA programming artifacts, such as clients or collaborating components. These port mechanisms specify different views and required interfaces that a component exposes to clients. Along with component attributes, these port mechanisms define the following capabilities of a component [Wang00]:

1. **Facets**: Facets, also known as *provided interfaces*, are interfaces that a component provides, yet which are not necessarily related to its supported interfaces via inheritance. Facets allow component to expose different views to its clients by providing different interfaces that can be invoked synchronously or asynchronously. For instance, the `Stock_Quote` interface in figure 2-23 provides a stock price querying capability to the component.

2. **Receptacles**: Before a component can delegate operations to other components, it must obtain the object reference to an instance of the other components it uses. In CCM, these references are "object connections" and the port names of these connections are called receptacles. Receptacles provide a standard way to specify interfaces required for the component to function correctly. In figure 2-23, the `Stock_Exchange` component uses the `SEC` (Securities and Exchange Commission) interface to function correctly. Using these receptacles, components may *connect* to other objects, including those of other components, and invoke operations upon those objects synchronously or asynchronously.

3. **Event sources/sinks**: Components can also interact by monitoring asynchronous events. These loosely coupled interactions, based on the Observer pattern are commonly used in distributed applications. A component declares its interest to publish or subscribe to events by specifying *event sources* and *event sinks* in its definition. For example, the `Stock_Exchange` component can be an event sink that processes `Buy_Offers` and `Sell_Offers` events and it can be an event source that publishes `Price_Change` events.

4. **Attributes**: To enable component configuration, CCM extends the notion of *attributes* defined in the traditional CORBA object model. Attributes can be used by configuration tools to preset configuration values of a component. Unlike previous versions of CORBA, CCM allows operations that access and modify attribute values to raise exceptions. The component developer can use this feature to raise an exception if an attempt is made to change a configuration attribute after the system configuration has completed. As with previous versions of the CORBA specification, component developers must decide whether an attribute implementation is part of the transient or persistent state of the component.

These new port mechanisms significantly enhance component reusability when compared to the traditional CORBA object model. For instance, an existing component can be replaced by a new component that extends the original component definition by adding new interfaces *without affecting existing clients* of the component. Moreover, new clients can check whether a component provides a certain interface by using the CCM `Navigation` interface, which enumerates all facets provided by a component. In addition, since CCM allows the *binding* of several unrelated interfaces with a component implementation entity, clients need not have explicit knowledge of a component's implementation details to access the alternative interfaces that it offers.

2.2.2.5 Development and Run-time Support Mechanisms

[Wang00] CCM addresses a significant weakness in CORBA specifications prior to version 3.0 by defining common techniques to implement CORBA servers. CCM extends the CORBA IDL to support components. Component developers use IDL definitions to specify the operations a component supports, just as object interfaces are defined in the traditional CORBA object model. Components can be deployed in component servers that have no advance knowledge of how to configure and instantiate these deployed components. Therefore, components need generic interfaces to assist component servers that install and manage them. CCM components can interact with external entities, such as services provided by an ORB, other components, or clients via *ports*, which can be enumerated using the introspection mechanism. Ports enable standard configuration mechanisms to modify component configurations. Figure 2-24 shows how the CCM port mechanism can be used to compose the components of our stock exchange example.

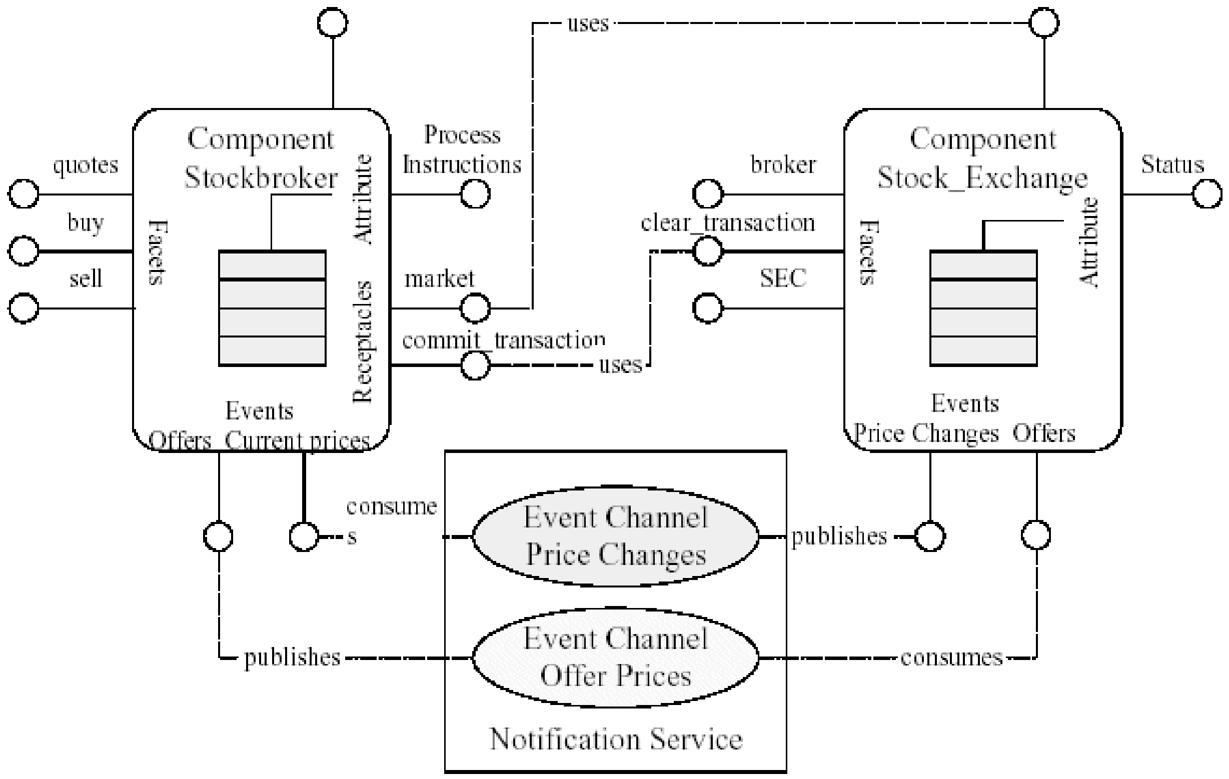


Figure 2-24: CCM component interaction through port mechanisms

CCM defines several interfaces to support the structure and functionality of components. Many of these interfaces can be generated automatically via tools supplied by CCM Providers. Moreover, life cycle management and the state management implementations can be factored out and reused. The CORBA Component Implementation Framework (CIF) is designed to shield component developers from these tedious tasks by automating common component implementation activities.

Many business applications use components to model “real world” entities, such as employees, bank accounts and stockbrokers. These entities may persist over time and are often represented as database entries. Components with persistent state are mapped to a persistent data store that can be used to reconstitute component state whenever the component instance is activated. For example, when a bank account component is instantiated, the CCM component model implementation is able to reconstitute the previous status of the account from a database. The CIF defines a set of APIs that manage the persistent state of components and construct the implementation of a software component.

CCM defines a declarative language, the Component Implementation Definition Language (CIDL), to describe implementations and persistent state of components and component homes. CIF uses the CIDL descriptions to generate programming skeletons that automate core component behaviors, such as navigation, identity inquiries, activation, and state management. Component implementations depend upon the standard CORBA Portable Object Adapter (POA) to dispatch incoming client requests to their corresponding servants.

2.2.2.6 Related Technologies

[Wang00] CCM is modeled closely on the Enterprise Java Beans (EJB) specification. Unlike EJB, however, CCM uses the CORBA object model as its underlying object interoperability architecture and thus is not bound to a particular programming language. Since the two technologies are similar, CCM also defines the standard mappings between the two standards. Therefore, a CCM component can appear as an EJB bean to EJB clients, and an EJB bean can appear as a CCM component by using appropriate bridging techniques. CCM and CORBA are also related to the Microsoft DCOM family of middleware technologies. The CORBA specification defines a bridging mechanism between CORBA objects and DCOM components. However, unlike CORBA and EJB, DCOM is limited mostly to Microsoft platforms.

2.2.2.7 Summary

In our opinion, CCM successfully extends the CORBA object model with support for components. Commonly used CORBA services, such as transaction, security, persistent state and event notification services, are now integrated in a standard environment. The new port mechanisms significantly enhance component reusability because a component can easily be replaced with another component which supports the same interfaces. By using bridging techniques a CCM component can appear as an EJB bean and vice versa. This can be an advantage for incorporating CCM in already existing EJB based systems. Also the fact that CCM is based on the popular CORBA standard will most likely increase popularity of CCM, because it will be easier to integrate CCM in a system that already is based on CORBA. However the language-neutrality of CCM allows incorporation of CCM in heterogeneous systems.

A disadvantage with CCM may be that the specification is large and complex. It is also a new standard (released June 2002) so ORB providers have only started implementing the specification recently. It is therefore still hard to evaluate the quality and performance of CCM implementations.

Advantages:

- Extends the CORBA IDL to support components.
- Integration of commonly used CORBA services in a standard environment.
- Port mechanisms enhance component reusability.
- Language-neutral allows incorporation in heterogeneous systems.

Disadvantages:

- Large and complex specification.
- New standard, hence little practical experience is available.
- Limited tool support.

2.2.3 Reusable Asset Specification (RAS)

This chapter contains information from [RAS02].

Rational Software and the Reusable Asset Specification Consortium are developing a set of guidelines for the specification, development and management of reusable software assets.

A fundamental prerequisite for reuse is an industry agreement on a common way of describing reusable software assets. The Reusable Asset Specification presented here is Rational's contribution to reaching such agreement.

The objective of RAS is to provide a set of practical and specific guidelines on how to describe reusable software assets in order to:

- Improve communication between asset producers and consumers.
- Represent assets in software development tools.
- Improve asset management and exchange.

All of the above are prerequisites for making reuse possible and to improved software development predictability, higher software quality and faster time to market.

The description of an asset is divided into several sections; Overview, Classification, Solution and Usage. The Overview section has no direct relationship to standards. The Classification part is related to meta-data standards that seek agreements on how to tag information assets with descriptors for grouping, storage and retrieval. The Solution part is related to software specification standards and practices and in particular to the UML standard. The Usage part has no relationship to standards.

RAS suggests a set of commonly used meta-tags; however it does not require that these specific tags must be used. In other words, the specification can work with any set of tags as long as they are agreed upon by the software community.

RAS uses UML in three ways:

1. It uses UML to describe itself (to describe asset description structure and semantics).
2. It uses UML for specifying the asset content.
3. It identifies a small set of UML extensions to precisely define properties of different asset categories and their variability.

RAS assumes that UML is used as the major means for specifying the content of a reusable asset.

RAS does not restrict the use of UML. In other words, whatever can be specified in UML, can be included in the content part of an asset description as long as the parameterization rules defined by the Reusable Asset Specification are followed. Hence, emergence of new UML profiles and domain specific UML extensions will benefit the Reusable Asset Specification.

2.2.3.1 Software Asset

A reusable software asset is a software artifact or a set of related artifacts that have been created or harvested with an explicit purpose of applying it repeatedly in subsequent development efforts. An asset has a description of the ways in which it should be used and applied. Artifacts are the result of software development activities and may reside as files on a file system. The asset package combines and brings together a set of related artifacts that together provide a solution to a problem.

Asset Categories

Software assets can be of different granularity; offer different degrees of customization; suite different phases in a software development lifecycle. There are three dimensions to assets:

1. Variability.
2. Granularity.
3. Articulation.

These can be used to discuss different categories or types of assets. A coarse interpretation and relative positioning of these types in software lingo is shown in the figure.

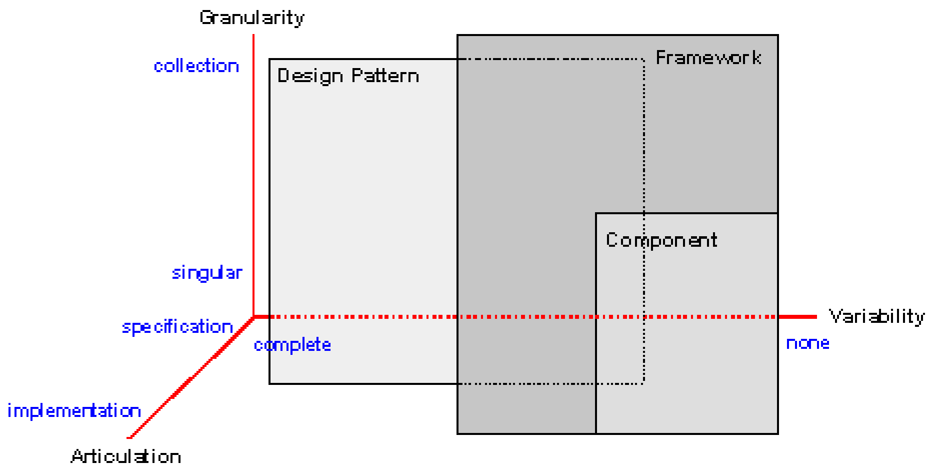


Figure 2-25: Classification of assets

Variability

The Variability dimension defines how customizable an asset is. Along this dimension assets range from templates, with few or none concrete elements, to components, with concrete elements with interfaces.

Granularity

The Granularity captures the size and complexity of an asset. Although size and complexity are relative, we can identify assets which are small collections of elements and assets which are large collaborations of smaller assets.

Articulation

The Articulation dimension follows the natural progression from requirements to designs to implementation. The above figure shows different articulations of a framework, from requirements, through design to implementation.

2.2.3.2 Asset Structure

Logical Asset Structure

An asset is a collection of artifacts, which are organized in a way that allows the asset to be analyzed, understood and applied. The Asset Structure Domain Model below represents a logical view of an asset.

Logical Asset Structure Domain Model

An asset is logically structured, containing the following sections: Overview, Classification, Solution, and Usage. This is captured in the Asset Structure Domain Model diagram below.

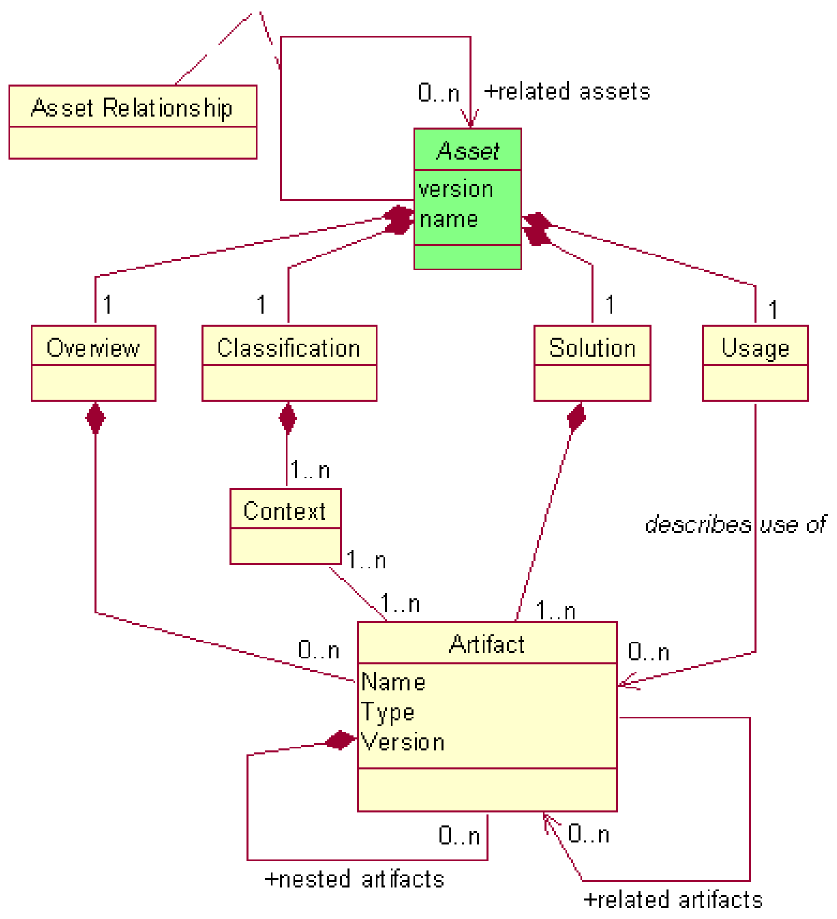


Figure 2-26: Logical Asset Structure Domain Model

Overview Section

The Overview section should contain text that would appear in a product catalog. It provides a general abstract of the asset, describing the problems the asset solves and the nature of the solution it provides. This is the top-level artifact in the asset and is the entry point or starting place when getting to know an asset.

This section can be perused by the casual observer and should contain text that will permit this. Below is a list of the kinds of documents one could expect in this section:

- *Asset Overview*: abstract of the asset, the problem it solves and the solution it provides.
- *Intent & Motivation*: description of the driving factors and intended purpose of the asset.
- *Problem Description*: description of the problem for which the asset provides a solution.
- *Applicability*: description of the intended targets for which the asset may be applied.
- *Related Assets*: overview of assets that may be related to an asset.
- *Standards*: any standards that the asset may submit to or be compliant with.
- *Support*: anticipated support issues regarding the asset.

Classification Section

The Classification Section has a set of tags or descriptors for classifying the asset. These descriptors are often organized as classification schemas. This section also describes the contexts in which the asset may be applied such as domain, development, test, and deployment contexts.

The model below describes the nature of the relationship between Assets and Classification Schemas. The model says that an Asset has one Classification section, which has a collection of Descriptor Groups. Any given Descriptor Group may conform to a Classification Schema.

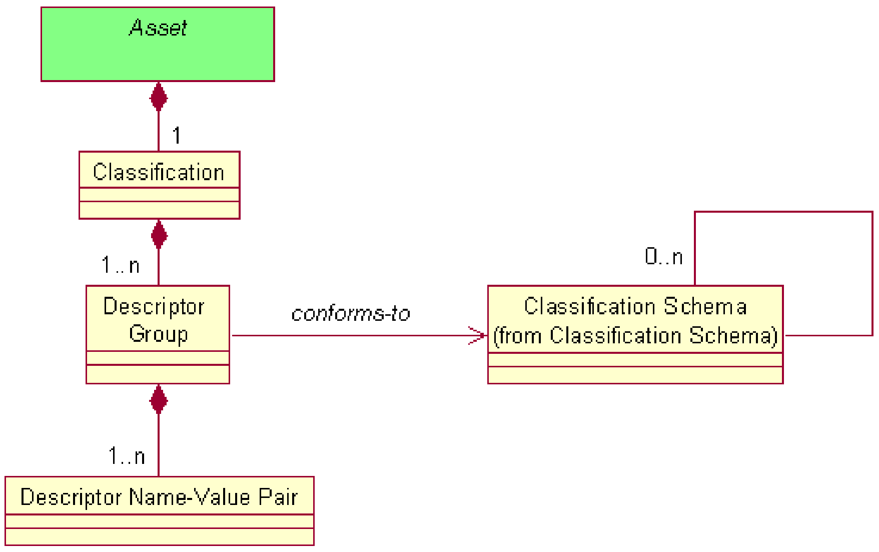

Asset
1
Classification
1..n
Descriptor Group
conforms-to
0..n
Classification Schema (from Classification Schema)
1..n
Descriptor Name-Value Pair

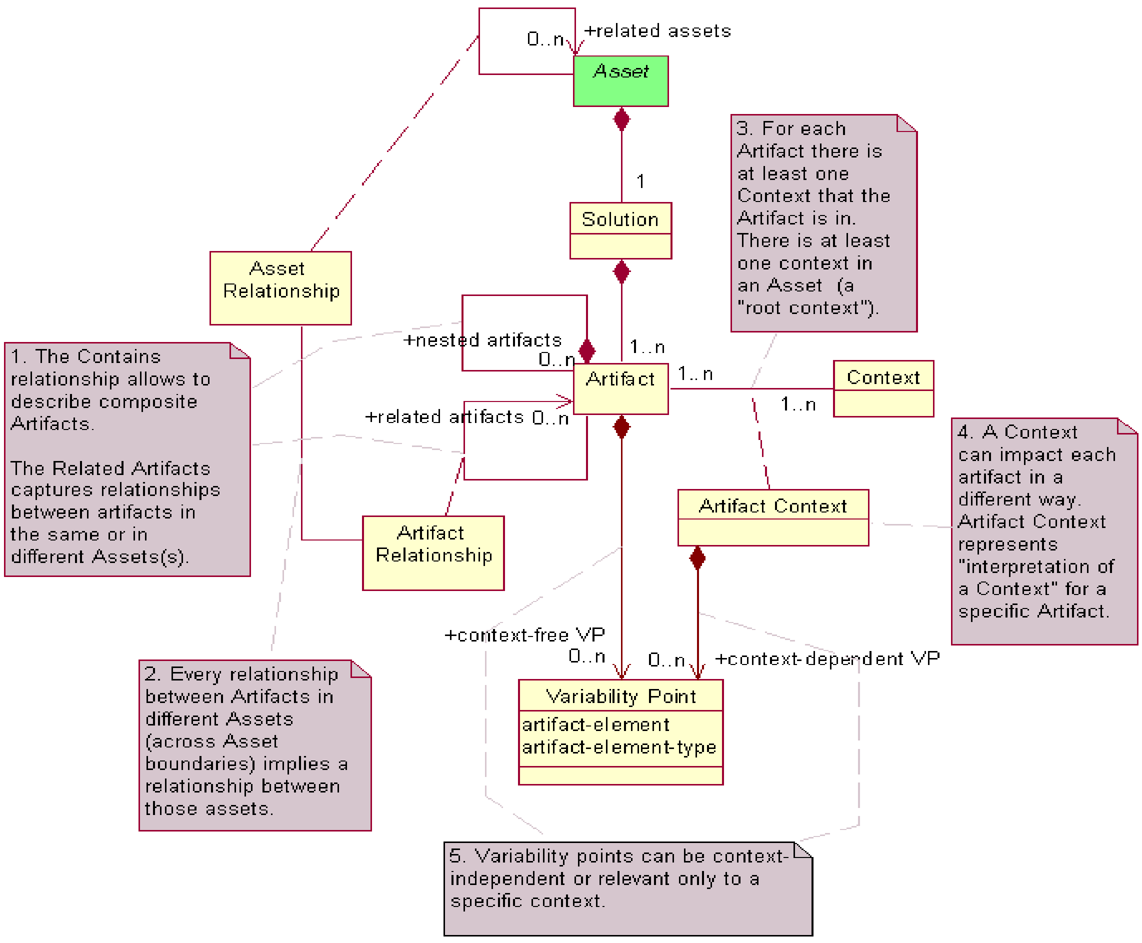
Figure 2-27: Classification section model

Classification Schemas may be created and defined in industry externally to any given project. In the model below the *Descriptor Group* and the *Descriptor Name-Value Pair* classes are part of the Asset Structure model. The model below describes the nature of *Classification Schema*s. It is anticipated that this model may be replaced by a specific vendor classification schemas and search engines.

Solution Section

The *Asset's* Solution section has the artifacts that are the models, files, documents, source code and components. These artifacts generally comprise the solution to a specific problem. The figure below illustrates the nature of the *Solution* section. The model says that an Asset has one and only one *Solution* section. The *Solution* section has a collection of *Artifacts*. *Artifacts* may contain or may reference other *Artifacts*. If an *Artifact* has a relationship with an *Artifact* in another *Asset*, it creates an *Asset Relationship*.

All *Artifacts* in an *Asset* have at least one *Context*, known as the *Root Context*. This identifies the starting point for the *Artifacts*, whether that is on a file system or in a model. An *Artifact* may be relevant to a specific *Context*. Each *Artifact* may have zero or more Variability Points. Some *Variability Points* may be dependent on a certain *Context*.



+related assets
0..n
Asset
1
Solution
3. For each Artifact there is at least one Context that the Artifact is in. There is at least one context in an Asset (a "root context").
Asset Relationship
+nested artifacts
0..n
1..n
Artifact
1..n
Context
1. The Contains relationship allows to describe composite Artifacts.
The Related Artifacts captures relationships between artifacts in the same or in different Assets(s).
+related artifacts 0..n
1..n
4. A Context can impact each artifact in a different way. Artifact Context represents "interpretation of a Context" for a specific Artifact.
Artifact Relationship
Artifact Context
+context-free VP
0..n
0..n
+context-dependent VP
2. Every relationship between Artifacts in different Assets (across Asset boundaries) implies a relationship between those assets.
Variability Point
artifact-element
artifact-element-type
5. Variability points can be context-independent or relevant only to a specific context.

Figure 2-28: Solution model

Usage Section

The Usage section provides the asset consumer with the directions and steps to approach and reuse the asset. The quality of an asset depends to a large extent on the quality of this section. This section describes the intersection of the sets of activities and the items on which those activities are relevant, when applying an asset.

The model below says that an *Asset* has one and only one *Usage* section. The major elements of the *Usage* section are *Activities* and *Variability Points*. Also, the model below describes the relationships of those two classes to *Artifacts* and *Context*.

Activities

The activities for applying an asset can be described in three basic scenarios:

For any given asset there may be a set of activities for applying the asset. For any given asset there may be a set of contexts for which the asset may be applied; and for each of those contexts there may be a set of activities that are relevant when the asset is being applied in a given context. For any given artifact there may be a set of activities for applying the artifact - both within a context and outside of a context.

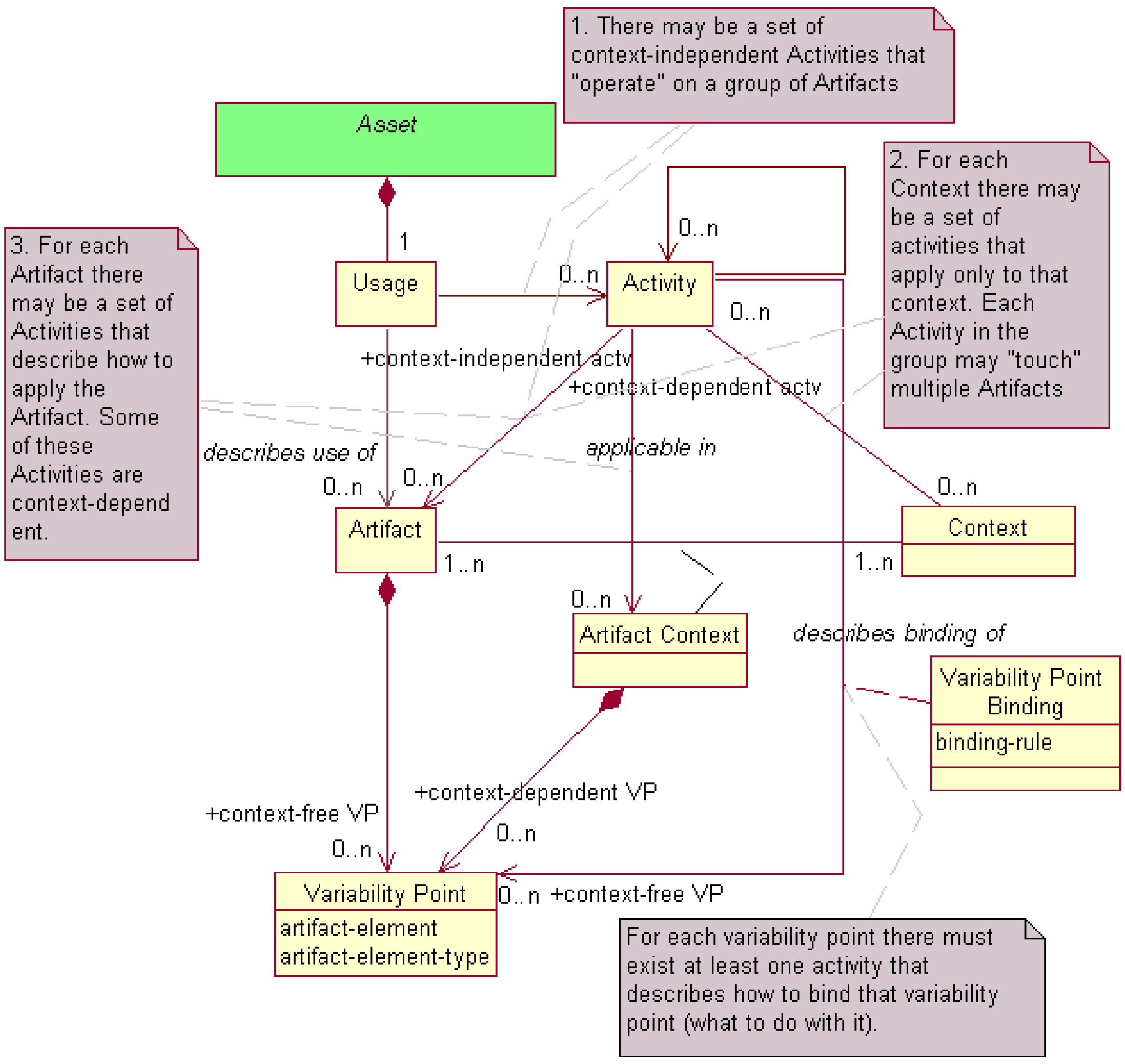

1. There may be a set of context-independent Activities that "operate" on a group of Artifacts
Asset
2. For each Context there may be a set of activities that apply only to that context. Each Activity in the group may "touch" multiple Artifacts
3. For each Artifact there may be a set of Activities that describe how to apply the Artifact. Some of these Activities are context-dependent.
Usage
Activity
1
0..n
0..n
0..n
+context-independent actv
+context-dependent actv
describes use of
applicable in
0..n
0..n
0..n
Artifact
Context
1..n
1..n
0..n
Artifact Context
describes binding of
Variability Point Binding
binding-rule
+context-dependent VP
+context-free VP
0..n
0..n
Variability Point
artifact-element
artifact-element-type
0..n +context-free VP
For each variability point there must exist at least one activity that describes how to bind that variability point (what to do with it).

Figure 2-29: Usage Model

<u>Variability Points</u>
Activities in the *Usage* section contain the binding rules for each *Variability Point*. These points are described for each *Artifact*, as necessary. However, the nature of the action that is to be performed is described in the *Activity*. The *Variability Point* binding-rule may also be dependent on the *Context* in which the *Artifact's Variability Point* is being filled.

Assets Relationships

Assets may refer to other assets. For instance, a framework may refer to components. Each of these assets may be self-standing, independent, yet related assets.

The figure below demonstrates this relationship, where larger assets may *logically* contain subordinate, independent assets. The green asset refers to the yellow asset. Notice the ***blue*** line connecting a green asset Artifact to a yellow asset Artifact. Typically, there is some relationship between the Asset's artifacts that creates or requires these kinds of asset relationships.

The asset contains and owns the artifacts within its scope. This is an ownership "by value." This means that if the asset is deleted - then the artifacts contained within that asset are deleted, unless some steps are taken to loosen this semantics. To reiterate, an asset may logically contain another asset, called containment by reference.

Asset relationships may occur at the artifact level and therefore cause a relationship to exist between assets. Or asset relationships may occur at the asset level and not to any particular artifact within the assets involved.

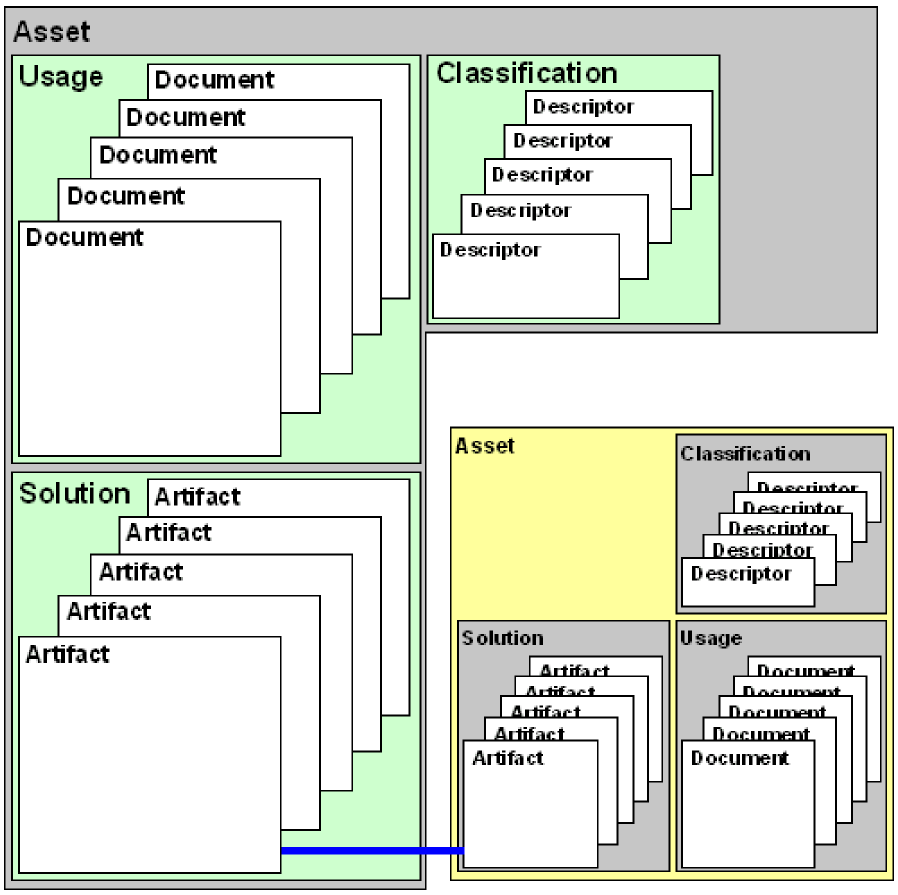


Figure 2-30: Assets Relationship

Physical Asset Layout

The artifacts and descriptors of an asset are logically grouped into several sections including Overview, Classification, Usage, and Solution. However, the actual location of these artifacts may be in different locations.

There are certainly many ways for assets to be represented physically. In this section we illustrate one way using XML Schemas and XML instance documents. The figure below illustrates the relationship between the Specification, the XML Schemas, and the actual XML asset instances.

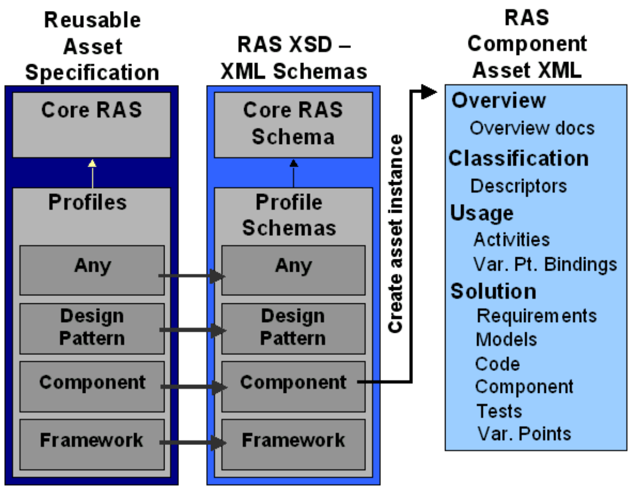


Figure 2-31: Asset Layout

Each section in the XML document may point to artifacts inside subdirectories of the same name. In this sense the XML file may be compared to a manifest file such as those in JAR files.

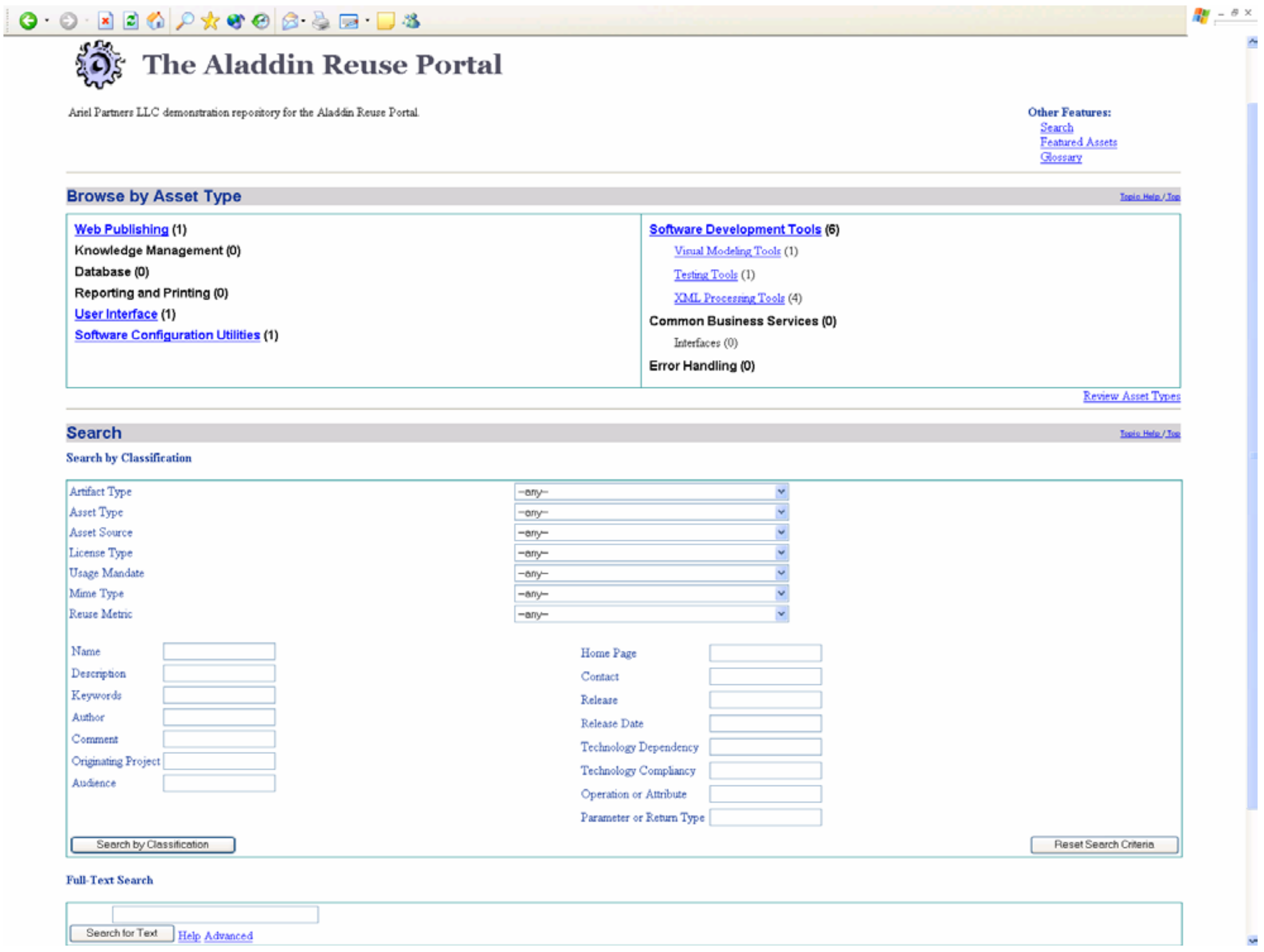
2.2.3.3 Tool Support for the Reusable Asset Specification

The Reusable Asset Specification (RAS) is a guideline and specification on how to make asset into packages which is easily identified and that can be inserted in a searchable database. This makes it possible to develop tools which searching, classification, distribution and inserting of assets possible. These tools appear as web portals, standalone applications and plug-ins and are becoming more and more available.

This chapter describes shortly the most common applications which support RAS.

Ariel Reuse Portal – Aladdin

Ariel [www.arielpartners.com] has developed a web portal called Aladdin Software Component Reuse Portal. This is a marketplace for distributing and searching of components within a company's intranet. All components are made into assets with RAS and inserted in a database and made available thorough Aladdin.



The Aladdin Reuse Portal

Ariel Partners LLC demonstration repository for the Aladdin Reuse Portal.

Other Features:
Search
Featured Assets
Glossary

Browse by Asset Type — Topic Help / Top

Web Publishing (1)	Software Development Tools (6)
Knowledge Management (0)	Visual Modeling Tools (1)
Database (0)	Testing Tools (1)
Reporting and Printing (0)	XML Processing Tools (4)
User Interface (1)	Common Business Services (0)
Software Configuration Utilities (1)	Interfaces (0)
	Error Handling (0)

Review Asset Types

Search — Topic Help / Top

Search by Classification

Artifact Type	--any--
Asset Type	--any--
Asset Source	--any--
License Type	--any--
Usage Mandate	--any--
Mime Type	--any--
Reuse Metric	--any--

Name		Home Page	
Description		Contact	
Keywords		Release	
Author		Release Date	
Comment		Technology Dependency	
Originating Project		Technology Compliancy	
Audience		Operation or Attribute	
		Parameter or Return Type	

Full-Text Search

Figure 2-32 - Aladdin Reuse Portal, screen shot

Rational Asset Browser

The Rational Asset Browser [Larsen02] is a prototype asset browser and a catalog manager. It comes in two different versions, a plug-in to Rational Rose and a standalone version, not requiring Rational Rose. This application enables you to browse, apply and manage any re-usable asset which conforms to the Reusable Asset Specification.

Rational Rose Asset Browser Plug-in

This asset browser is a plug-in for Rational Rose, but it is not supported by Rational and it is purpose is only for demonstration of how an asset browser could look like. The browser is also an asset packed after the RAS standard it self.

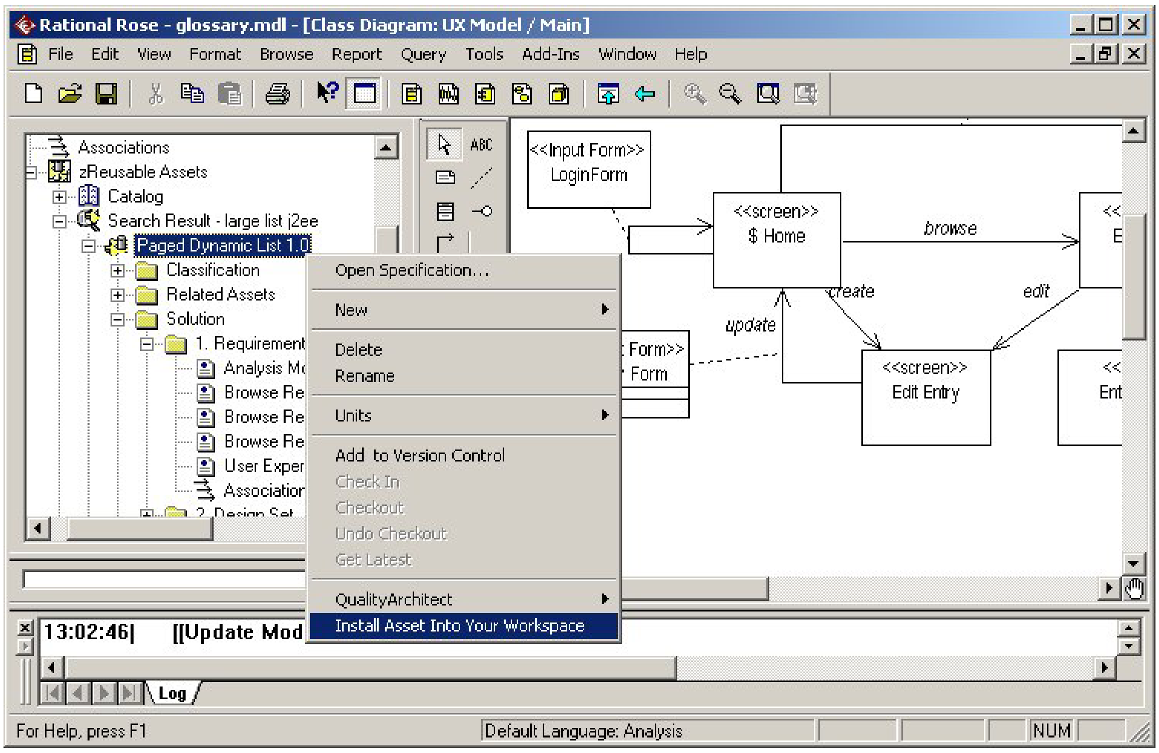


Figure 2-33: Inserting an asset into the Rose workspace

Rational Asset Browser

The Rational Asset Browser is a standalone tool for browsing and applying assets. This tool has no search abilities, so all assets browsed must be known to the user. The Asset Browser is an asset it self.

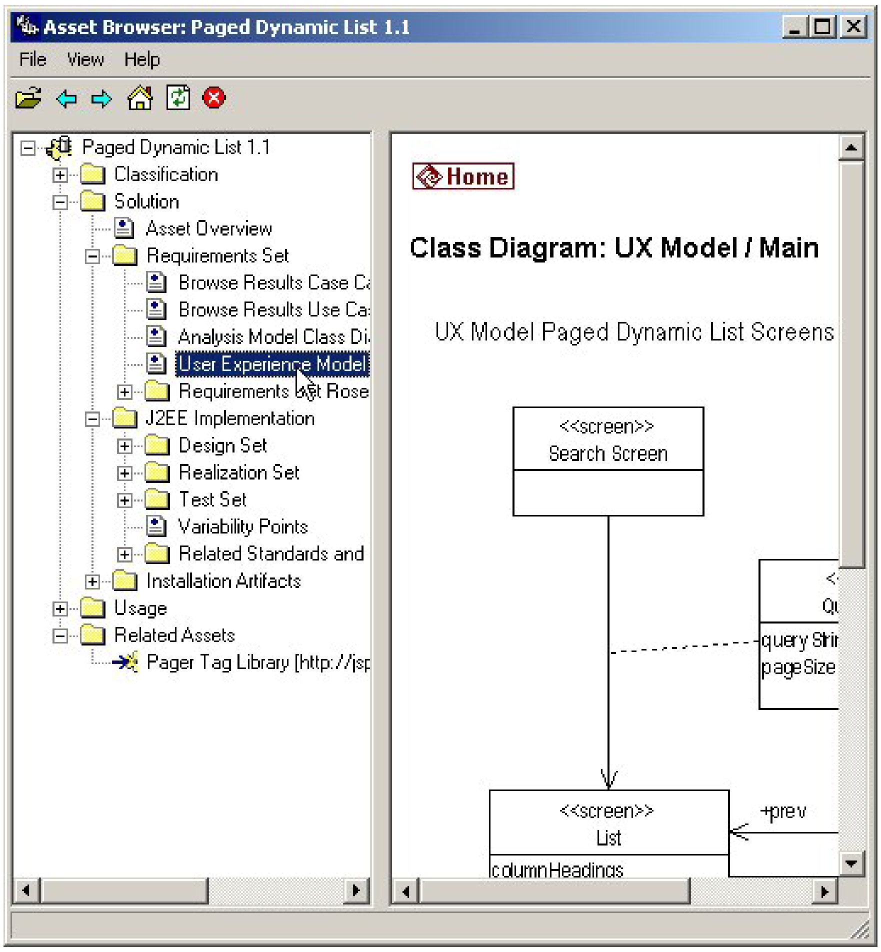


Figure 2-34: Rational Asset Browser standalone tool

Reusable Asset Mining Portal (RAMP)

Blueprint Technologies [www.blueprinttech.com] has developed the Reusable Asset Mining Portal (RAMP) which is a web-based tool for discovery and packaging of software assets.

RAMP includes a mining engine which Blueprint Technology claim is capable of automatic searching of directory trees, databases and websites. Blueprint Technology further claims that RAMP automatically packages and inserts this information as assets in the RAMP database, using the Reusable Asset Specification.

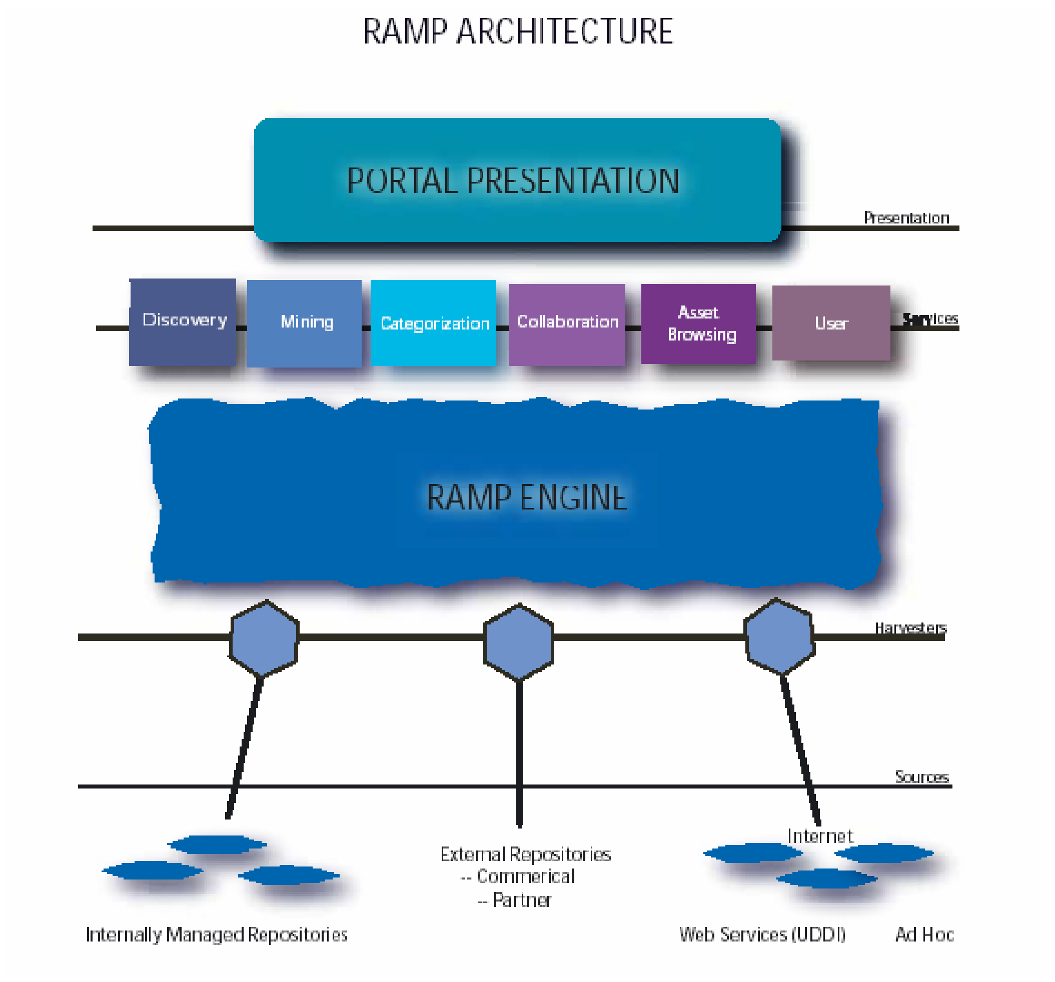


Figure 2-35: RAMP Architecture

Logidex

LogicLibrary [www.logiclibrary.com] has developed a software tool named Logidex. LogicLibrary claims that this tool is capable of packing and discovery of software assets and offers a catalog of these to its users.

LogicLibrary also offers professional consultant help on implementing web services, improving enterprise reuse, component-based development, enterprise architectures and better understanding of the evolution of software asset inventory.

2.2.3.4 Summary

In our opinion, RAS provides a useful structure for formally specifying and organizing assets and their artifacts. The goal is to increase the anticipated user experience with regards to reuse and decrease the associated costs. Reducing the costs of reuse decrease the pay off threshold and strengthen the business value proposition of reuse.

RAS does not formally specify the metrics to run your reuse business, nor does it decrease the complexity of the reusable item. RAS still rely on skilled architects and engineers to prescribe a properly sized, reusable item.

Advantages:

- RAS does not prefer any specific tools and are meant to be tool-neutral.
- The description of assets is done using the standard UML from OMG. This makes RAS easy to understand and learn.
- By requiring a set of description attributes for each asset RAS makes it easy to collect assets in a searchable database.

Disadvantages:

- RAS does not give any guidelines to how to identify already existing assets in the development process.
- The specification does not say anything about how to model assets for best possible reusable potential.

2.3 Evaluation of State-of-the-art

This chapter contains an evaluation of the state-of-the-art processes and standards we have examined in this report. First we provide a comparison of processes, comparing KobrA, COMET and RUP. We then compare the state-of-the-art standards for component specification, comparing UML 1.x, UML 2.0, CCM and RAS.

2.3.1 Comparison of Processes

Our comparison is based on [Solberg97], [Boertien01], [Atkinson00] and [Atkinson01]. [Boertien01] did not include KobrA in the survey and [Stoj01] only included RUP. Our contribution has been to complete the comparison and we have added 4 other classification criteria. In the table below stands ✓ for yes and ✗ stands for no.

Method	KobrA	COMET	RUP
Background [1]	Academic (/Industry)	Academic	Industry
Type of process [2]	Development (see text on next page)	Development and management	Development and management
Usage of methodology [1]	Industry partners are probably the only users of today	Only in user case within OBOE project	Adopted by some major players
Process support [1]	Not prescriptive. Supports a range of methods	Milestones; Guidelines	Milestones; Guidelines; templates; and tool mentors
Applied in what phases of software development [1]	Requirements, analysis & design and implementation	Analysis & design and implementation	Requirements, analysis & design and implementation
Component identification process [3]	✓ (including variant functionality for identifying reusable components)	✓	✗ (no specific activities on identification)
Units of reuse [1]	Components; frameworks	Components; patterns; frameworks	Software components
Modeling techniques used [1/2]	Several UML diagramming techniques	Several UML diagramming techniques	UML and others
Tool support [1/2]	No dedicated support, standard UML tools can be used for modeling	No dedicated support, standard UML tools can be used for modeling	Extensive support from the Rational Suite
Components defined in [3/4]	KobrA Components (Komponents)	Business Objects (BO)	UML components
Encourages iterative development [3]	✓	✓	✓
Implementation platform [1]	Platform independent	Business Objects framework (BOF) on top of CORBA	Platform independent
Easily integrable with RUP [3]	✓	✗	N/A

Figure 2-36: Comparison chart over KobrA, COMET and RUP

[1] Classification criteria from [Boertien01]
[2] Classification criteria from [Stoj01]
[3] Our classification criteria
[4] To make the identification process possible, both KobrA and COMET includes more information (documentation) in addition to the plain UML diagrams. This is information is encapsulated in KobrA's Komponents and in COMET's BOs.

Figure 2-36 shows how KobrA and COMET relates to the RUP process. What seems obvious is the lack of component identification process in RUP. This is covered in both KobrA and COMET. RUP uses standard UML notation to define components.

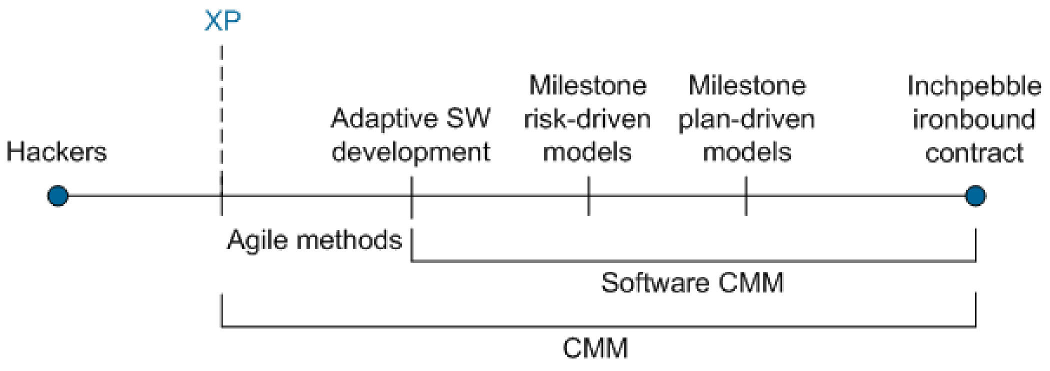

XP
Hackers
Adaptive SW development
Milestone risk-driven models
Milestone plan-driven models
Inchpebble ironbound contract
Agile methods
Software CMM
CMM

Figure 2-37: Boehms planning spectrum

Figure 2-37 illustrates Boehms planning spectrum as presented in [Boehm02]. Unplanned and undisciplined hacking occupies the extreme left, while micromanaged milestone planning, also known as inchpebble planning, occupies the extreme right. In figure 2-38 we use Boehms planning spectrum [Boehm02] to illustrate where we would place KobrA, COMET and RUP.

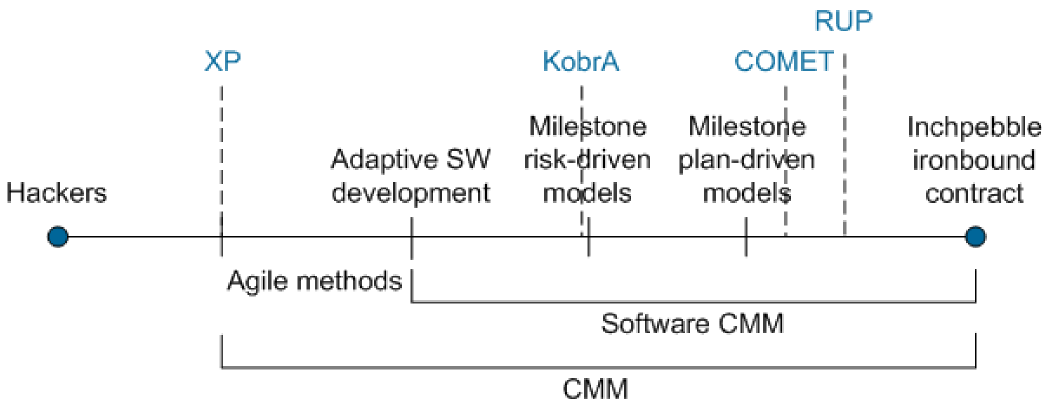

RUP
XP
KobrA
COMET
Hackers
Adaptive SW development
Milestone risk-driven models
Milestone plan-driven models
Inchpebble ironbound contract
Agile methods
Software CMM
CMM

Figure 2-38: Proposed placement of methods in Boehms planning spectrum

RUP and COMET are placed together left of the extreme inchpebble ironbound contract and to the right of plan-driven models. As shown in figure 2-36 in the column “Type of process” both RUP and COMET offers the managerial aspect of the development process. KobrA does this as well, but is not as specific in this aspect as RUP and COMET and is therefore listed as Development only. In the sense that KobrA defines concrete rules for structuring development artifacts, and systematic activities for developing them, it can be viewed as a “heavyweight” process. However, its emphasis on the incremental addition of features and the recursive elaboration of artifacts gives it some of the agility of so called “lightweight” (or agile) methods. The main distinction between KobrA and other methods that are typically viewed as being lightweight is its emphasis on modeling. Boehms planning spectrum has a strong focus on planning and not the modeling aspects. When seeing that KobrA is placed in between adaptive SW development and milestone risk-driven models one should keep in mind that the complete picture is a bit more complex. Placing software development methods in relation to each other in a one dimensional way is a simplification of the real world.

2.3.2 Comparison of Standards

The following table is our own comparison of UML 1.x, UML 2.0 and CCM. In the table stands ✓ for yes and × stands for no.

Standard	**UML 1.x**	**UML 2.0**	**CCM**	**RAS**
Type of standard	Modeling language for OO systems	Modeling language for OO and component- based systems	Framework for constructing component-based CORBA system	Specification of reusable assets
Support for Component-Based Development	Inadequate	✓	✓	✓
Specification of component in what part of design	Deployment diagram (late in design phase)	High level design (early in design phase)	High level design (early in design phase)	All parts of design phase
Component interaction	Interfaces	Ports	Ports	N/A
Tool support	Numerous tools exist	None[5]	Several tools exist	Plug-in for Rational Rose and stand-alone application
Platform independent	✓	✓	✓	✓

Figure 2-39: Comparison chart over UML 1.x, UML 2.0 and CCM

The most noticeable difference between UML 1.x and UML 2.0 is the inadequate support for Component-Based Development of UML 1.x. UML 2.0 and CCM support specification of components early in the design phase, while UML 1.x specifies components in the deployment diagram of the design model. The port mechanism of UML 2.0 and CCM for specifying required and provided interfaces offers flexible reuse and replacement of components.

CCM is a complete framework for building large-scale distributed component-based systems, while UML is only a modeling language for object oriented systems. The added support for component-based development of UML 2.0 allows modeling of component-based systems like CCM. We therefore think that UML 2.0 and CCM are complementary standards, i.e. UML 2.0 may be used to model a CCM based system.

2.3.3 Summary

In chapter 4 we use the state-of-the-art process and standards explored in this chapter in order to suggest improvements on the current practice at Ericsson. In this chapter we also mention the need for continuous process improvement in order to implement the suggested changes.

[5] The UML 2.0 standard is not yet finished, so no tools exist at this time.

3 Current Practice at Ericsson

This chapter contains our study of the current practice at Ericsson, Grimstad. We have looked at their adapted GSN RUP process, especially the potential for component identification in the requirements and analysis & design workflow. We then present a brief overview of the GSN architecture before we analyze the structure and practice of component specification in GSN UML model.

The word "practice" may be exaggerated; we have no special empirical evidence in what degree Ericsson really follows the GSN RUP process and their modeling practices.

3.1 About Ericsson

[Ericsson02] Ericsson is the largest supplier of mobile systems in the world. The world's 10 largest mobile operators are among Ericsson's customers and some 40% of all mobile calls are made through Ericsson systems. Ericsson provides total solutions covering everything from systems and applications to services and core technology for mobile handsets. With Sony Ericsson we also are a top supplier of complete mobile multi-media products. Ericsson has been active worldwide since 1876 and the company has today around 71,700 employees in more than 140 countries. The headquarters is located in Stockholm, Sweden.

3.2 Ericsson GSN

GPRS (General Packet Radio Service) allows large amounts of data to be sent over mobile networks at speeds three to four times greater than conventional GSM systems. Because the data is sent in packets, GPRS enables operators to offer customers simultaneous, data-rich services, such as multimedia messaging, gaming, entertainment, and news, over their mobile phones.

Providing telecom services today and for the future, Ericsson has expanded the current GSM network with the new 3rd Generation (3G) system using UMTS standard technology for fast data transmission. The vast majority of the customers will still use the GSM standard for some years to come, forcing Ericsson to run a network where GSM and UMTS can coexist. The core of this network is the GSN (GPRS Support Node) system.

3.3 GSN RUP Adaptation

3.3.1 GSN RUP Overview

Ericsson AS uses a tailored or adapted version of RUP, called GSN RUP. Figure 3-1 shows the GSN RUP process map with phases and workflows. The information in this chapter contains edited information and figures from [GSNRUP02].

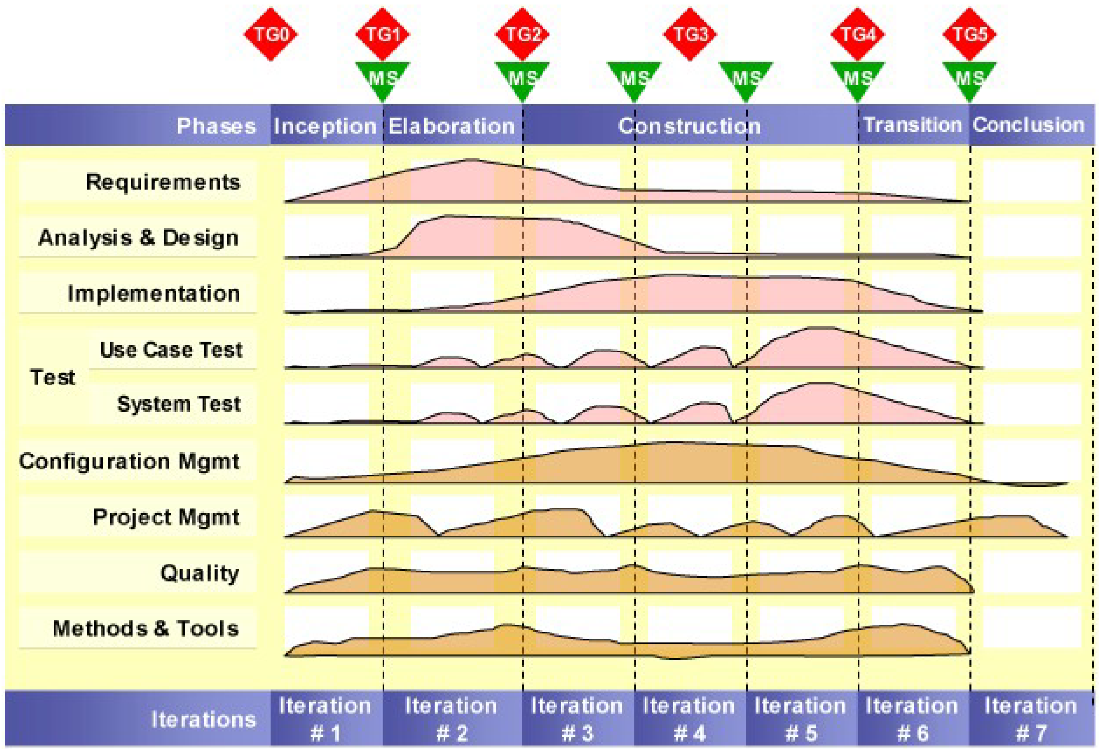


Figure 3-1: GSN RUP process map

3.3.2 Requirements Workflow

The information in this chapter is intended to give the reader an overview of the requirement workflow in GSN RUP. The emphasis and focus is on component identification (most important is the chapter on the Feature Impact Study).

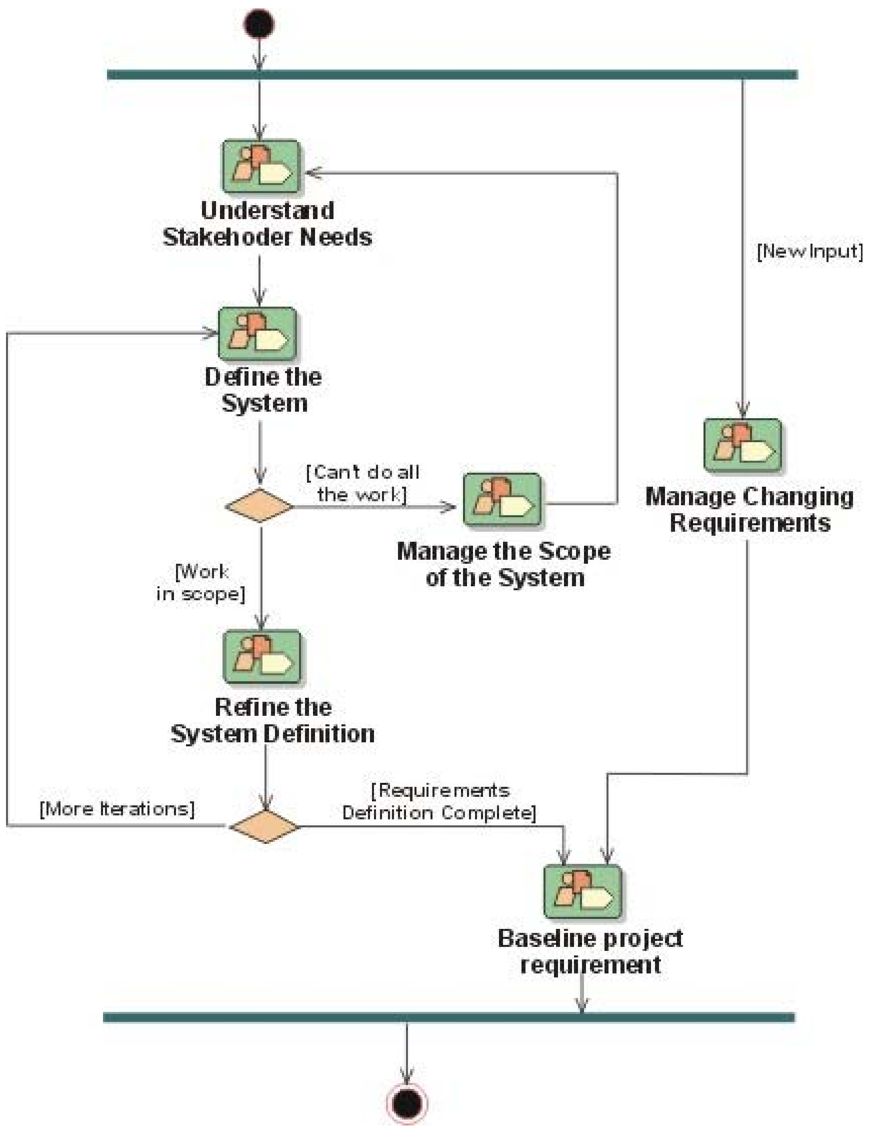

Understand Stakehoder Needs
[New Input]
Define the System
[Can't do all the work]
Manage the Scope of the System
Manage Changing Requirements
[Work in scope]
Refine the System Definition
[More Iterations]
[Requirements Definition Complete]
Baseline project requirement

Figure 3-2: Overview of the Requirement Workflow

The prime goal of Requirement Management is to develop a clear understanding of what the product should do. This workflow is an adaptation of the standard RUP defined Requirement Management workflow with the ambition to help Ericsson convert to a more "feature-by-feature delivering software factory" compared to the previous "each project delivers a complete release" idea.

Standard RUP defines the artifacts "Stakeholders request" and "Vision" as placeholders for this kind of information, but for the current implementation of RUP the "ARS" (Application Requirements Specification) will replace such artifacts. The information contained therein would typically be "slogan-level" requirements.

The ARS will then provide some input for the preceding workflow activity, "Define the system", in which an effort is made to understand what the system should do on a "rough" level and spread this knowledge to all involved project members. This is the first break–down of requirements and it involves both functional and non-functional requirements.

With the requirements broken down, it is possible to perform a prioritization of the requirements. This is done within the workflow detail "Manage the Scope of the System". An evaluation is then done on whether the current scope for a project (or iteration), as defined through requirement attributes, is realistic or if a re-prioritization is needed (that is, feedback to ARS level).

In "Refine the System Definition" all requirements are further refined, i.e. use case Specifications are detailed and Supplementary Specifications completed. This is the final break-down of requirements ensuring all details are covered and should be followed by a revisit to "Manage the Scope of the system" as new information affecting scope may have been encountered.

"Manage Changing Requirements" is about handling the requirement changes that will always enter a project over time, ensuring that these are incorporated into the requirement base in a controlled way. It is also about structuring the use case model to make requirement changes more manageable.

The workflow detail "Baseline project requirements" is finally added for handling the baselining of product requirements to project requirements, the reason being that the GSN product will evolve through several projects (and releases/drops) and the requirement scope must be frozen for each. For a detailed description see the "GSN Requirements Management plan".

Understand Stakeholders Needs

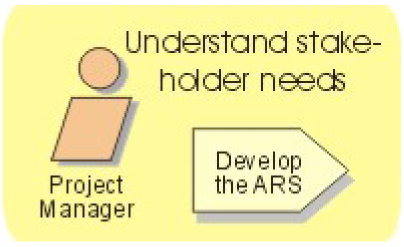


Figure 3-3: Overview of the workflow detail "Understand stakeholders' needs"

The Requirement Management workflow starts with the activity "Understand stakeholders needs", where the focus is on grasping what current and future needs Ericsson customers (internal as well as external) may have. The purpose of this workflow detail is to collect and elicit information from stakeholders to the project. The collected stakeholder requests in the ARS will be used as primary input to defining the use-case model, use-cases and supplementary specifications. Typically, this activity is mainly performed during iterations in the inception and elaboration phases.

Traditionally the ARS has been used for scoping a project, a scope that is frozen i.e. at the TG2 decision (normally just before the construction phase starts). With the introduction of requirement attributes, the selection of ARS requirements, i.e. scoping may be done through attribute setting.

Define the System

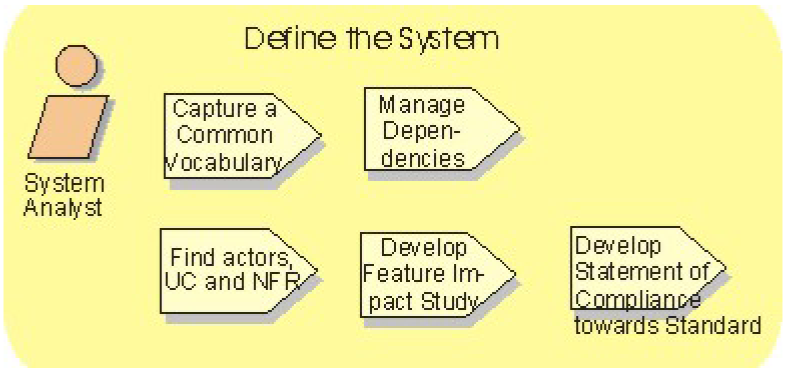

Define the System
System Analyst
Capture a Common Vocabulary
Manage Depen-dencies
Find actors, UC and NFR
Develop Feature Im-pact Study
Develop Statement of Compliance towards Standard

Figure 3-4: Overview of the workflow detail "Define the System"

The main purpose of this workflow detail is to align the project team in their understanding of the system requirements. This is done by performing a first analysis and break-down of ARS requirements in the activity Find actors, Use Cases and Non-functional Requirements. The output from these activities is then used in the Develop Feature Impact Study activity to secure that all the required resources are available in the project.

The activity Develop Feature Impact Study is not found in the standard RUP. The reason for including both this activity is that it is important for deciding if the project has the resources to mange the scope presented in the ARS regarding functional and non-functional requirements.

The standard RUP does not handle non-functional requirements in this workflow detail, but since the non-functional requirements prove to have at least as much impact on the resulting GSN product as the functional requirements have, the GSN RUP Adaptation has chosen to do that.

Capture a Common Vocabulary

The main purpose of this workflow detail is to define a common vocabulary that can be used in all textual descriptions of the system, especially in use-case descriptions. This is done by first defining a common vocabulary using the most common terms in the problem domain. Then use the common vocabulary in all textual descriptions of the system, especially in use-case descriptions, to avoid misunderstandings among project members about the use and meaning of terms. If there should be a conflict, the terms defined here or in other standards have priority over other meanings of the same terms. When this is done a formal review is needed if the glossary is intended for external use.

Manage Dependencies

In this workflow detail attributes and traceability of project requirements is managed to assist in managing the scope of the project and manage changing requirements. The following steps are performed to do this:

- **Choose Requirements Attributes**
  Choosing attributes is normally performed in the initial stage of a project. Later the defined requirement attribute set may have to be updated, i.e. as a result of some attributes not being used or the project requesting a new attribute.

- **Using Requirements Attributes**
  Using requirement attributes will be helpful in order to assess the impact of changes of scope, test failures, requirement changes and to verify that the system does only what it is intended to do.

- **Establish Traceability**
  Traceability furthermore enhances the ability to detect the impact of changing requirements, thus pointing out what requirement parts are influenced. I.e. an ARS level requirement will impact on several Use Cases and perhaps some non-functional requirements.

Find Actors, Use Cases and Non-Functional Requirements

The workflow detail "Find actors, use cases and non-functional requirements" outlines the functionality of the product and the system and defines what will be handled by the system and what will be handled outside the system. Who and what will interact with the system is also defined before the model is divided into packages with actors and use cases. Diagrams of the use-case model are created and optionally a survey of the use-case model is developed. Every requirement in the ARS shall be considered in this activity to secure that a proper FIS can be made, but the level of details in the Use Case sketches and the non-functional requirements shall only be good enough to do a Feature Impact Study (FIS).

Develop Feature Impact Study

General about FIS

The purpose of a FIS is to describe how a feature (requirement objects from an ARS) influences a certain system Issue. It also includes an estimation of the amount of work that is needed to carry out the implementation.

The FIS is intended to be a container document, where one to a large degree only reference the information in other artifacts, hence providing a complete picture of the work needed to be done in order to successfully implement a feature. Every feature or system improvement that is considered for a system should be handled in a FIS before it is accepted to be a part of the project scope. The FIS is also used as the primary analysis artifact for Change Requests.

Before starting this activity, the steps "Find actors" and "Find Use Cases" in the activity Find Actors and Use Cases have to be completed and used as input to this activity.

The FIS should consist of minimum one or preferable several proposals on how to implement a feature or a part of a feature. The FIS document should consist of the information needed to make a decision if it will/can be implement. The different areas to consider are described in the FIS guideline. The costs in man hours are essential to make a decision for implementation. This estimation should be based on the information claimed in the FIS work.

Users of FIS

Those who need information from the FIS are:

- Product management - to verify that the requirements are adequate and have been correctly interpreted.

- Project management - to plan resources for the design work to be performed after the System Analysis.

- System management – to verify that the proposed solution is in line with the wanted architecture of the system.
- Designers and testers- to establish a picture of the influence on functions/features, products and documentation.

FIS Stages

The preparation of a FIS is divided into three stages:

- Stage 0 or Pre TG0 (Responsible: Product and System Management)
- Stage 1 or TG0 – TG1 (Responsible: Pre-study project)
- Stage 2 or TG1 – TG2 (Responsible: Development project)

The level of detail in the FIS should be determined from one case to another, and certainly depend on what stage the document is in.

Normally less than 100-200 man-hours should be used for each FIS (for all three stages).

Some elements such as System Architecture Description, Design Patterns, Design Rules, Analysis Model and Design Model is worked on during all three stages, but should if possible be completed by stage 2. These elements are therefore not included in the following subchapters describing the different stages. (Most of the tasks described in each stage are resulting in information gathered in chapter 6 (- Technical Solution) of a FIS.)

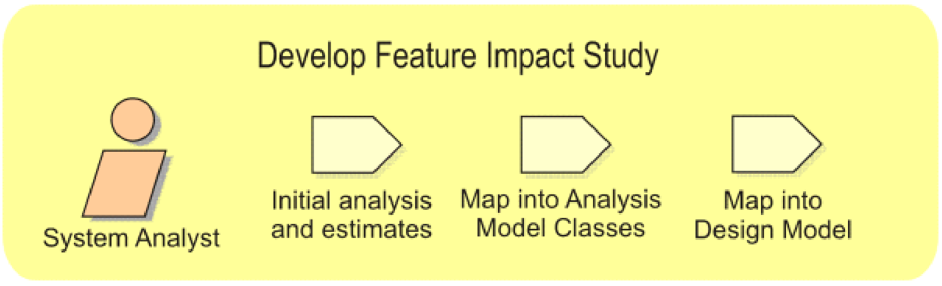

Develop Feature Impact Study
System Analyst
Initial analysis and estimates
Map into Analysis Model Classes
Map into Design Model

Figure 3-5: Proposed overview of the workflow detail "Develop Feature Impact Study"

<u>Stage 0 (Initial analysis and estimates)</u>

In the first stage it is necessary to clarify the requirements. This is done by first listing the Application Requirement Specification (ARS), Use-Case sketches and supplementary specifications that are input to this FIS. Also when applicable, Change Requests that have influence on this FIS should be listed. A description of how the requirements have been interpreted should be made.

The implementation is broken down and costs are estimated on node level and the changes in interfaces between the system components (Application, Business Specific, etc.) are described briefly.

<u>Stage 1 (Map into Analysis Model Classes)</u>

In the second stage the implementation is further broken down, and mapped it onto the Analysis Model classes (UML). This is done while taking special care to identify the need for new Analysis classes, and/or major changes in the interaction between the existing classes.

The cost estimates from Stage 0 are revised, and the impact on each system component is estimated. If applicable any new/influenced interfaces between the system components is describe or if possible modeled, and any possible incompatibility issues must by identified.

Stage 2 (Map into Design Model)
In the third and last stage the implementation is broken down again, this time mapped onto the Design Model (UML), taking special care to identify the need for new Design classes, and/or major changes in the interaction between the existing classes. In this stage the implementation is modeled down to (but not deeper) a level where the implementation gets dependant upon a particular design base (subsystem level).

The cost estimates from the previous stage are revised again, and the impact on each subsystem component is estimated. Any new/influenced interfaces between the subsystem components, taking care to identifying any possible incompatibility issues are describe (or if possible modeled).

Alternative proposals may be given in case of uncertainty of the best alternative. The preferred alternative is placed first, and once a decision has been made as to what alternative is selected, the remaining alternatives are removed. Then a brief overview of what must be added, amended, removed or replaced compared to the Design Base in order to obtain the new feature or system improvement is added.

A brief description of the implementation is then made as a process showing the actions within the relevant products, showing sequences and interactions. It is encouraged that diagrams and figures is used to facilitate understanding. All new features should be structured and the responsibility for each function (or functionality part) and the interaction between them must be described.

Develop Statement of Compliance

In this activity a Statement of Compliance (SoC) is created. It is a formal description of what parts of a Standard (ETSI and others) that will be supported in the final product and for some cases what Ericsson decided changes there may be.

Manage Changing Requirements

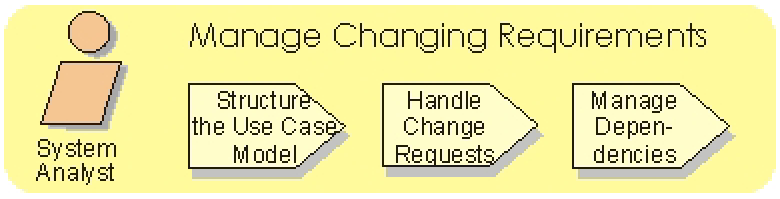


Figure 3-6: Overview of the workflow detail "Manage Changing Requirements"

The purpose of this workflow detail is to:

- Structure the Use Case Model.
- Evaluate formally submitted change requests and determine their impact on the existing requirement set.
- Set up appropriate requirements attributes and traceability.

- Formally verify that the results of the Requirements workflow conform to the customer's view of the system.

Changes to requirements naturally impact the models produced in the Analysis & Design workflow, as well as the test model created as part of the test workflow. Traceability relationships between requirements identified in the Manage Dependency activity of this workflow and others, such as Analysis & Design and Test, is the key to understanding these impacts.

Another important concept is the tracking of requirement history. By capturing the nature and rationale of requirements changes, reviewers (in this case the role is played by anyone on the software project team whose work is affected by the change) receive the information needed to respond to the change properly.

Manage the Scope

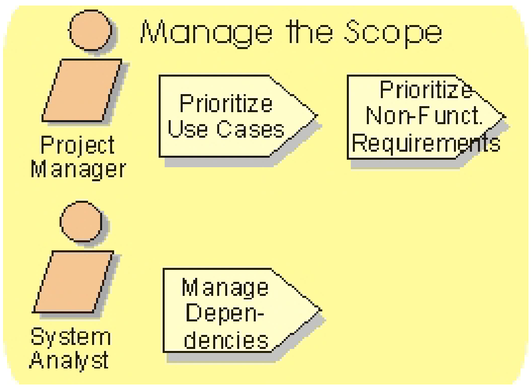


Figure 3-7: Overview of the workflow detail "Manage the Scope"

The main purposes of this workflow detail are:

- Prioritize and refine input to the selection of features and requirements that are to be included in the current and future iterations.
- Define the set of use cases (or scenarios) that represent some significant, central functionality.

This will affect the Iteration Plan that is the responsibility of the Project Manager. The System Architecture Description (SAD) will also be affected, which implies that the A&D Roles must be incorporated in this work.

An input to "Managing the Scope of the System" workflow detail not seen in other workflow details of the requirements workflow is the Iteration Plan, developed in parallel by project and development management.

The Iteration Plan defines the number and frequency of iterations planned for the release, as well as the prioritized Use Cases (per Iteration).

The scope of the project defined in "Managing the Scope of the System" will have a significant impact on the Iteration Plan as the highest risk elements within scope will be planned for early iterations.

Refine System Definition

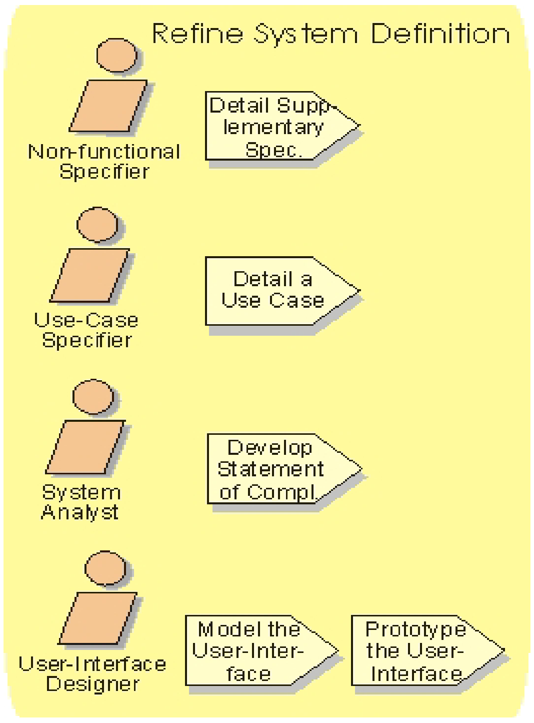


Figure 3-8: Overview of the workflow detail "Refine System Definition"

The purpose of this workflow detail is to further refine the requirements in order to:

- Describe the use case's flow of events in detail.
- Detail Supplementary Specifications.
- Develop Statement of Compliance – towards standards.
- Model and prototype the user interface (if there is any).

Refining the System begins with use cases outlined, actors described at least briefly, and a revised understanding of project scope reflected in re-prioritized features in the Feature Impact Study that is believed to be achievable by fairly firm budgets and dates. The output of this workflow is more in-depth understanding of system functionality expressed in detailed use cases, revised and detailed Supplementary Specifications and Statement of Compliance, as well as user-interface elements (if there are any).

Baseline Project Requirement

The purpose of this workflow detail is mainly to create baseline project requirements from the product requirements. The project will then design and implement these requirements.

3.3.3 Analysis & Design Workflow

The first part of this chapter describes the Analysis & Design workflow in GSN RUP with focus on component identification, followed by a description of the GSN UML model with focus on modeling of components in the GSN architecture.

This workflow spans mainly over the phases Elaboration and the first two iterations in the Construction phase. The main artifacts here are analysis model and the design model. The analysis model is created in the Elaboration phase, and is updated in the Construction phase as the structure of the model is updated. The software architect is responsible for this artifact.

The design model primarily describes the architecture, but is also used for analysis during the elaboration phase. It is kept consistent with the UC model and the implementation model. The software architect is responsible for this artifact, but designers are responsible for packages, classes and so on.

The activities in this workflow can be described with the following flow chart.

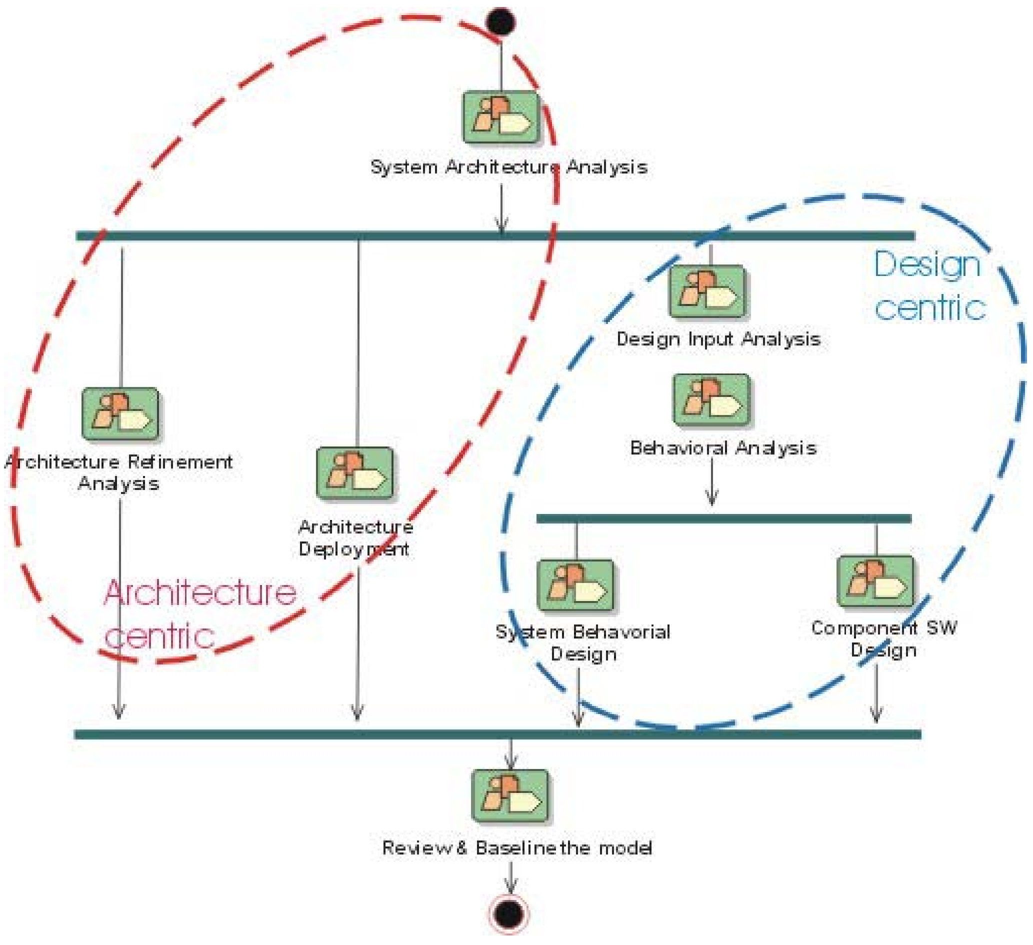


Figure 3-9: Analysis & Design workflow

There are two main categories of A&D activities. As the GSN projects will not build a new system from scratch (there is an existing design base), the architecture centric activities will focus on improving and evolving the current system architecture to meet future requirements. The design centric activities, on the other hand, will focus on implementing new functionality in the existing system.

Architecture Centric Activities

These activities are for the most concerned with the analysis model, but also interacting with the design model. Architects have the main responsibility.

System Architecture Analysis

The purpose of this workflow detail is to analyze the architectural needs based on the requirements. This will lead to work in different sub-packages in the system architecture (i.e. the analysis model) in the Rose model. This package contains the system architecture for the Ericsson GSN products.

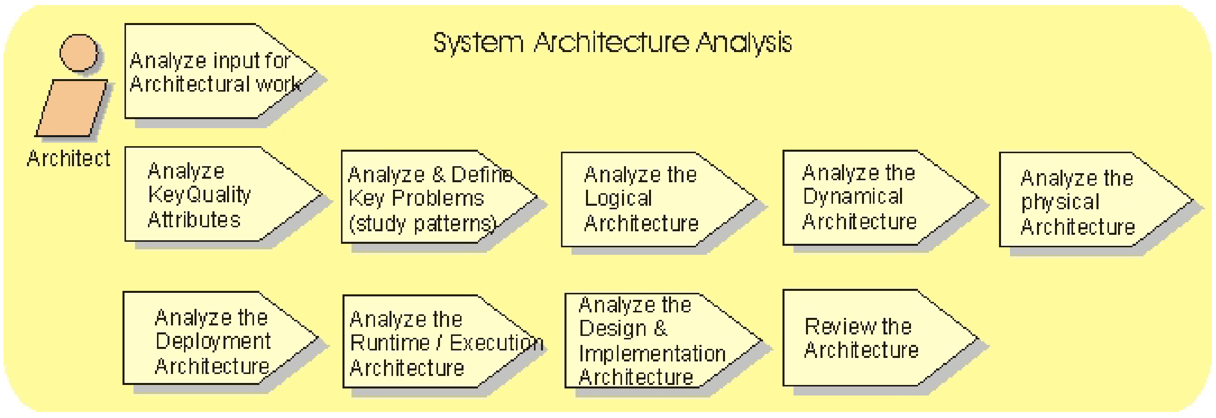

System Architecture Analysis
Architect
Analyze input for Architectural work
Analyze KeyQuality Attributes
Analyze & Define Key Problems (study patterns)
Analyze the Logical Architecture
Analyze the Dynamical Architecture
Analyze the physical Architecture
Analyze the Deployment Architecture
Analyze the Runtime / Execution Architecture
Analyze the Design & Implementation Architecture
Review the Architecture

Figure 3-10: Overview of the workflow detail "System Architecture Analysis Workflow"

Analyze Input for Architectural Work

The purpose of this activity is to understand the requirements on a high level and their effect on the system. The analysis shall give a first indication on the impact of the new requirements on the system. If the requirements forces major change in the existing interfaces, this must be captured at this early stage in the architectural work. To analyze the functional and non-functional requirements effect on the system a workshop is often initiated with participants representing both the functional and the non-functional requirements i.e. System Analysts and the Product management.

Analyze Key Quality Attributes

The purpose of this activity is to identify and analyze the attributes and properties of the system that will have a dominating influence on the architecture. The Architect is responsible. Some choices are made on what will be the key quality attributes, i.e. scalability, maintainability or availability, and document these choices in the Software Architecture Description (SAD). Subsequently each of the key quality attributes are separately analyzed to find out the consequences of choosing these attributes as steering for the system architecture.

Analyze & Define Key Problems (study patterns)

The purpose of this activity is to identify the key problems and analyze and study existing architectural- and design patterns for a possible solution.

Analyze the Logical Architecture

The purpose of this activity is to define the logical structure of the system based on the strategies and concepts defined in the SAD and the architecturally significant Use Cases. The architect shall organize the structure into areas of interest (domains) that can be analyzed separately. The architect shall identify the major domain terms and concepts and describe them in the SAD, and then further break down each domain in analysis classes with purpose and responsibility and model their dependencies and associations in the logical view in the system architecture.

Analyze the Dynamical Architecture

The purpose of this activity is to identify the required key requirements as well as other scenarios and analyze the necessary principles for interaction.

Analyze the Physical Architecture

The purpose of this activity is to analyze the physical aspects of the architecture i.e. the hardware; the physical components that build the GSN node and the distribution of logic onto the physical nodes. The responsible role is Architect. GSN RUP does not describe how to develop the hardware. The physical architecture view is rarely modeled in Rose, instead other tools more suited for this kind of illustrative drawings (i.e. PowerPoint, Visio etc...) are used and the drawings are linked into the Rose model.

Analyze the Deployment Architecture

The purpose of this activity is to define the principles for deployment of logic to hardware. The Architect shall draw up the framework for software deployment and high level realization of this framework.

Analyze the Run-time / Execution Architecture

The purpose of this activity is to analyze the run-time aspects of the system i.e. the need for concurrency. Especially performance and scalability, but also availability qualities shall be considered.

Analyze the Design & Implementation Architecture

The purpose of this activity is to analyze the results from several of the other activities in the System Architecture Analysis workflow group and with that knowledge as a base, *identify and define different types of design components at least to an Ericsson subsystem level*. The responsible role is Architect. Also the Architect shall define some architecturally significant lower level design components (function blocks). In this activity the scope of each component is defined by its responsibility. The subsystems are defined in the design model. In the identification of design components, the Architect must have an organizational experience and the overall architectural competence/knowledge is very important.

Review the Behavioral Design

The purpose of this activity is to assure the quality of the design and get approval of the design choices from the required authorities. This is done by performing model reviews and inspections according to the review process.

Architecture Refinement Analysis

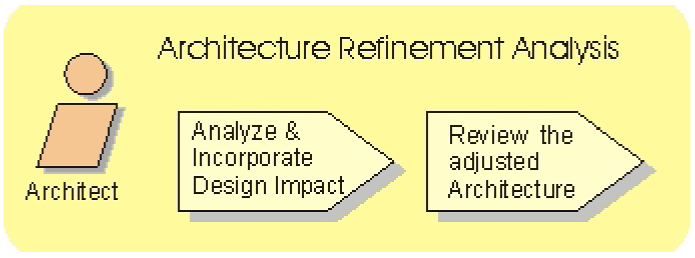

Architecture Refinement Analysis
Architect
Analyze & Incorporate Design Impact
Review the adjusted Architecture

Figure 3-11: Overview of the workflow detail "Architecture Refinement Analysis"

The purpose of this workflow detail is to make changes to the architecture based on changed requirements and changed perception of what the requirements mean and to allow new design that was not foreseen at an earlier stage.

Architecture Deployment

The purpose of this workflow detail is to distribute knowledge about the architecture to all parts of the organization through training courses and direct support to designers and design & implementation teams.

Design Centric Activities

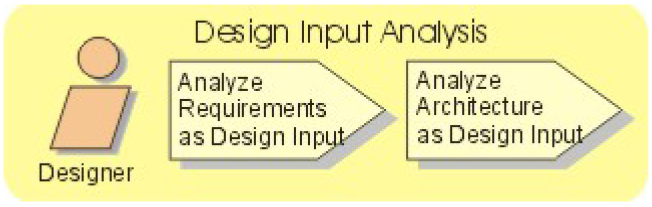

Design Input Analysis
Designer
Analyze Requirements as Design Input
Analyze Architecture as Design Input

Figure 3-12: Overview of the workflow detail "Design Input Analysis"

These activities are for the most concerned with the design model, but also interacting with the analysis model. Designers have the main responsibility.

Design Input Analysis

The purpose of this workflow detail is to obtain a clear picture of all the input to the design work including all requirements, and to make sure that all participants understand the architecture and other issues that constrain the design.

Analyze Requirements as Design Input

The purpose of this activity is to understand the requirements that are input to the design work and their impact on the system. To analyze the requirements a workshop is often initiated with participants representing both the functional and the non-functional requirements.

Analyze Architecture as Design Input

The purpose of this activity is to understand the important concepts in the architecture and their effect on the system. The activity is teamwork between the designers and the architects. The analysis shall give the designer the framework in which the design will be produced. A way for the designers to analyze the architecture is to arrange seminars/training courses held by architects. It is important that the designers are aware of the architecture and the artifacts that describe it, so that they can make good design choices.

Behavioral Analysis

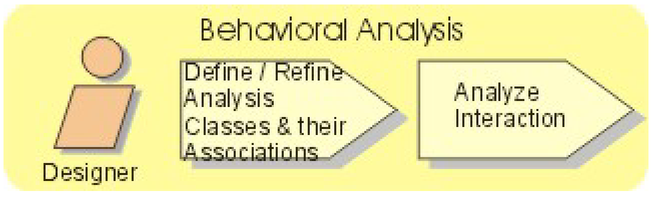


Figure 3-13: Overview of the workflow detail "Behavioral Analysis"

The purpose of this workflow detail is to analyze the behavior that the requirements specify.

Define/Refine Analysis Classes & their Associations

The purpose of this activity is to capture analysis classes that are appropriate for realizing the requirements. This is done by assigning responsibilities to existing and new classes. In this activity the architects and the designers will systematically go through the analysis already done by the architects but also analyze new aspects of the new functionality. It is important for the architects and designers to understand the behavioral needs required by the system architecture and the system requirements and to assign responsibilities to the appropriate analysis classes. The result can be to merge, split or modify the existing classes or to add new classes. Workshops may be arranged to support the design teams.

Analyze Interaction

The purpose of this activity is to analyze how the defined/refined analysis classes interact to realize the requirements. The behavioral analysis is a team activity and an experienced architect or designer should participate. They must understand the behavioral needs required by the system architecture and the system requirements by analyzing the interactions between the analysis classes. They identify scenarios in the requirements and model them in temporary interaction diagrams with the analysis classes in the design model.

System Behavioral Design

The purpose of this workflow detail is to distribute the behavior to subsystems and blocks and define interfaces and data types (in the IDL files) that allow them to communicate.

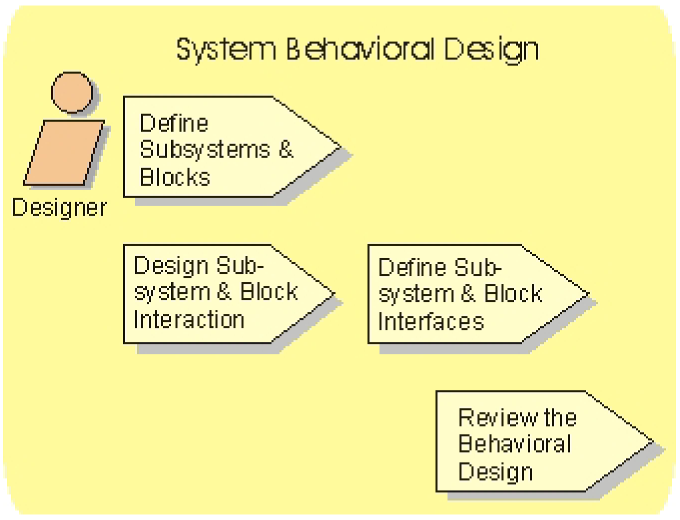

System Behavioral Design
Designer
Define Subsystems & Blocks
Design Sub-system & Block Interaction
Define Sub-system & Block Interfaces
Review the Behavioral Design

Figure 3-14: Overview of the workflow detail "System Behavioral Design"

Define Subsystems and Blocks

The purpose of this activity is to define Subsystems and Blocks that are appropriate for realizing the analysis classes used in the Behavioral Analysis and to assign responsibilities to them. This implies finding subsystems and blocks that already has the needed responsibilities or assigns new responsibilities to suitable subsystems and blocks. It is important that each subsystem and block has a coherent responsibility and that they are not too tightly coupled. The design may result in new subsystems and blocks being created or splitting or merging of already existing subsystems and blocks. This is done first on subsystem level and then on block level. The resulting components are documented in a class diagram showing the subsystems and blocks and their dependencies.

Design Subsystem and Block Interaction

The purpose of this activity is to analyze how subsystems and blocks interact to realize the behavior described in the Behavioral Analysis. Interfaces with operations are invented and modeled in the design model. Component behavior is modeled using interaction diagrams, activity diagrams and state diagrams.

Define Subsystem and Block Interfaces

The purpose of this activity is to specify the formal interfaces of subsystems and blocks in full detail in IDL-modules. The designer shall define interfaces and data types directly in the IDL-code in the IDL-modules. Each IDL-module must have the same name as the corresponding Rose package and each operation in the defined interfaces must have the same name as the operations in the corresponding interface classes in the design model. The interfaces are defined at block level and often exported out to subsystem level.

Review the Behavioral Design

The purpose of this activity is to assure the quality of the design and get approval of the design choices from the required authorities. This is done by performing model reviews and inspections according to the review process.

Component Software Design

The purpose of this workflow detail is to design the internal behavior of a component. A component is either a subsystem or a block.

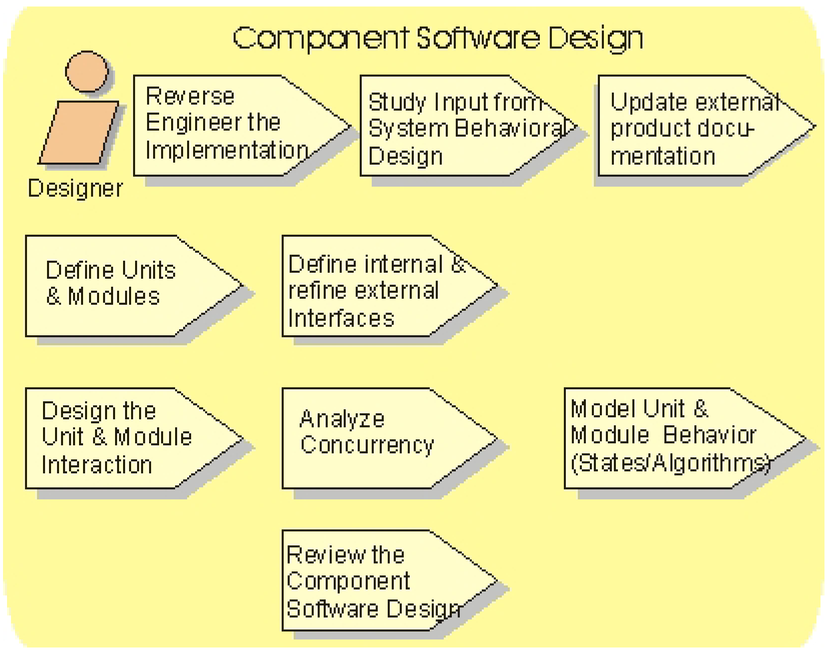

Component Software Design
Designer
Reverse Engineer the Implementation
Study Input from System Behavioral Design
Update external product documentation
Define Units & Modules
Define internal & refine external Interfaces
Design the Unit & Module Interaction
Analyze Concurrency
Model Unit & Module Behavior (States/Algorithms)
Review the Component Software Design

Figure 3-15: Overview of the workflow detail "Component Software Design"

Define Units and Modules

The purpose of this activity is to define which units and modules that are needed to realize the component. It is recommended to separate the component responsibility into clearly defined areas that can be assigned to units. A unit is made up of a number of modules. There must be at least one module for each unit. Each module is a source code file written in Erlang or C.

Define Internal & Refine External Interfaces

Change or add new external interfaces to subsystems or blocks and define needed interfaces between the units.

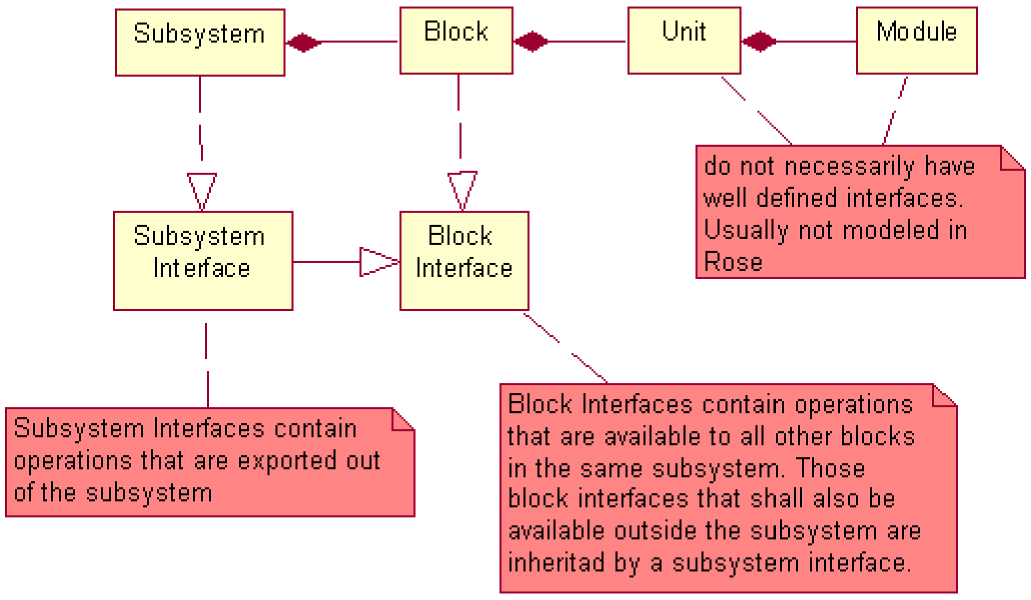

Subsystem
Block
Unit
Module
Subsystem Interface
Block Interface
do not necessarily have well defined interfaces. Usually not modeled in Rose
Subsystem Interfaces contain operations that are exported out of the subsystem
Block Interfaces contain operations that are available to all other blocks in the same subsystem. Those block interfaces that shall also be available outside the subsystem are inheritad by a subsystem interface.

Figure 3-16: GSN component types and their interfaces

Design the Unit and Module Interaction

The purpose of this activity is to define how the units and modules interact to fulfill the defined responsibilities. This is done by drawing interaction diagrams using elements defined in the activity Define Units & Modules, showing a selection of the internal sequences that realize the behavior.

Analyze Concurrency

The purpose of this activity is to consciously and in a planned way analyze the need for parallel execution and concurrency in the units and modules.

Model Design Element Behavior (States / Algorithms)

The purpose of this activity is to model the behavior of single design elements. The behavior and the conditions for alternative behaviors are detailed through state models and activity diagrams.

3.3.4 Component Identification in GSN RUP

Each project has their own architect group that is responsible for identifying components. The members of this group are usually experienced and have extensive knowledge of the reusable components that already exist. This experience is invaluable to Ericsson as they are not using a database for storing information on reusable components, but rely on experience for identifying reusable components.

3.3.4.1 Component Definition

The GSN RUP definition of component [GSNRUP02]: "*A component represents a piece of software code (source, binary or executable), or a file containing information (for example, a startup file or a Read Me file). A component can also be an aggregate of other components, for example, an application consisting of several executables*."

3.3.4.2 Requirements Workflow

The initial component identification activity is done in the activity “Define the System” and specifically in the sub activity “Develop the Feature Impact Study (FIS)”. In the FIS the work needed to implement a specific feature is mapped to the work that has to be done. To have a complete picture of how much work that has to be done it is necessary to identify which parts that have to be developed from scratch and what parts that can be reused (i.e. from the framework). Our proposed new GSN RUP figure (figure 3-5 on page 70) shows the three main activities that are performed in a FIS. All three phases are important for component identification as they are closely linked. The first phase called “Initial Analysis and Estimates” the costs on node level are estimated. In this phase a preliminary decision is made on what can and can not be reused. The second and third phase (“Map into Analysis Model Classes” and “Map into Design Model”) the implementation is further broken down, and mapped onto UML.

3.3.4.3 Analysis & Design Workflow

In our opinion, an important activity in the Analysis & Design workflow regarding the identification of components is the “Analyze the Logical Architecture” activity. Here the Architect breaks down each domain in analysis classes identifying purposes and responsibilities and models their dependencies and associations in the logical view in the system architecture. The identification of analysis classes is important because it is the foundation for the subsequent process of decomposing the system into components, and because it is mainly driven by requirements.

In the activity “Analyze the Design & Implementation Architecture” the Architect shall identify and define different types of design components at least to an Ericsson subsystem level. As mentioned in [GSNRUP02] it is important that the Architect must have an organizational experience and sufficient overall knowledge of the system architecture. This implies that the identification process is mainly based on the previous experience and knowledge of the Architect.

Finally in the activity “Define Subsystems and Blocks” (see page 79) the Designer shall find subsystems and blocks that already have the needed responsibilities or assign new responsibilities to suitable subsystems and blocks. In this activity the Designer identifies the main design components. This process is also based on the previous experience of the Designer.

3.3.5 Product Line Engineering

It is the worker called Framework Responsible that has the overall responsibility for all that is added to a specific framework part (i.e. Capella, see chapter 3.4.5). He works together with the project team and decides if anything should be changed or added. It is important to note that it is not this worker that takes these decisions alone, but together with the project-team. If there are cross platform reuse issues a joint team takes decisions according to an article written by Parastoo Mohagheghi and Reidar Conradi [Conradi01]: "*When the reuse is across products and organizations in Ericsson, a joint team of experts (called the Software Technical Board, SW TB) takes the decision regarding shared artifacts. The SW TB should address identification to verification of the reusable component and together with the involved organizations decide which organization owns this artifact (should handle the development and maintenance). Teams in different product areas support the SW TB*".

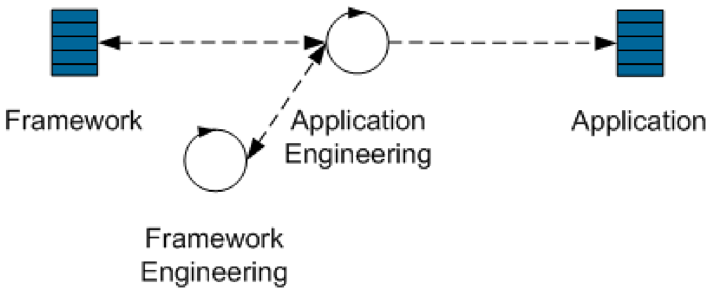


Figure 3-17: Product Line Engineering in GSN RUP as we see it

Changes to the framework and identification of reusable components are now performed in the application engineering activity only. This is described in more detail in chapter 3.3.4. There really does not exist any separate framework engineering activities; they are an indistinguishable part of the application engineering activity. In figure 3-17 this is illustrated by drawing framework engineering activity as a separate process that *only* interacts with the application engineering activity (see figure 2-4 on page 21 for notation explanation. To see an example on how product line engineering is done in KobrA see figure 2-5 on page 21). The framework engineering activity in GSN RUP is experienced based, and the Framework Responsible (see chapter 3.3.5) has few or no guidelines to rely on.

3.4 GSN Architecture

This chapter contains information from [GSNSAD02] and [GSNMOD02]. Each of the major telecom standards, GSM and UMTS includes a part for wireless packet data communication. These two parts have a lot in common and are evolving together. Ericsson has a GPRS/UMTS packet network realizing the wireless packet data communication. GPRS (General Packet Radio Service) is a new non-voice value added service that allows information to be sent and received across a mobile telephone network.

3.4.1 The Core Network

A common packet domain Core Network is used for both GSM and UMTS. This common Core Network provides packet-switched (PS) services and is designed to support several quality-of-service levels in order to allow efficient transfer of real-time and non-real-time traffic.

A GPRS Support Node (GSN) contains functionality required to support GPRS functionality for GSM and/or UMTS. The SGSN and GGSN nodes constitute the Ericsson GSN system. Figure 3-18 shows a simplified version of the GSN system and the rest of the core network.

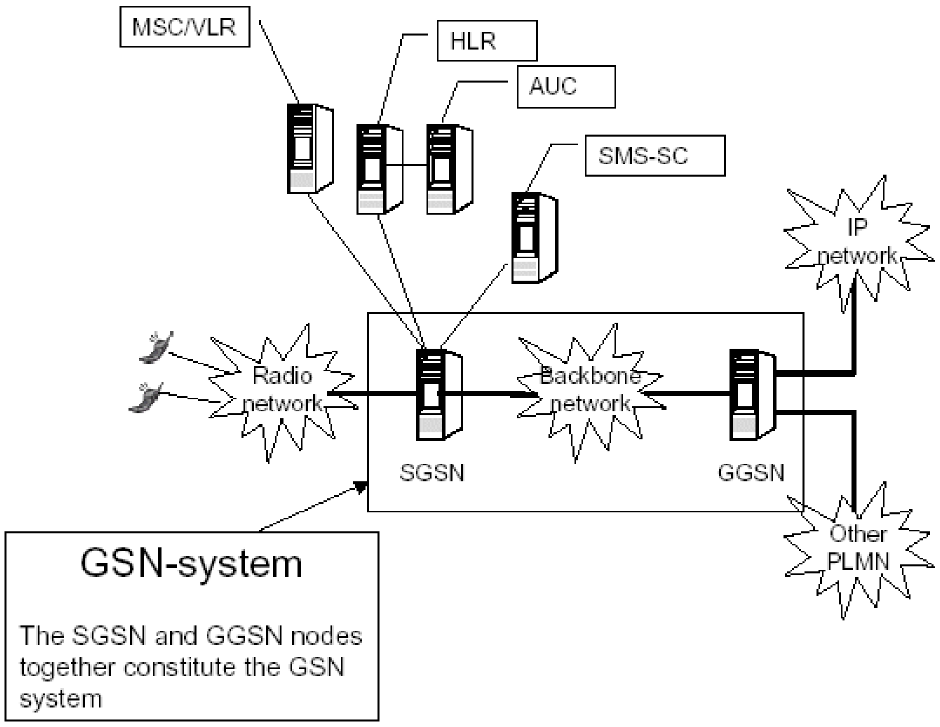


Figure 3-18: Overview of the Ericsson packet-data core network

SGSN keeps track of the individual mobile stations (cell phones and other wireless devices) location and performs the necessary security functions. In addition to being connected to the GSM base station and to the UMTS Radio Access Network, some network service nodes are also connected. Examples of these are the Home Location Register (HLR), responsible for providing the SGSN node with GSM and UMTS subscriber information. GGSN provides internetworking with external packet switch networks. It is connected to SGSN via an IP-based backbone network.

3.4.2 GSN System Family

Originally there existed a whole family of GSN based on the same architecture idea.

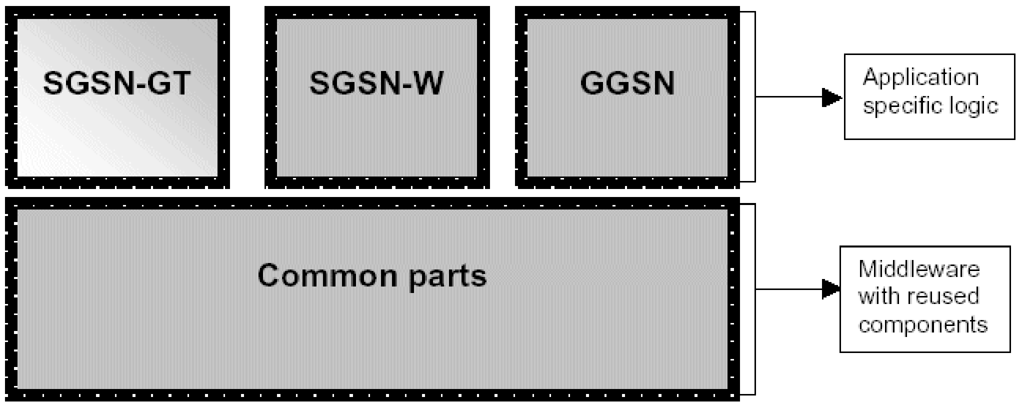


Figure 3-19: The GSN System family

- Common parts: Implementing the common/generic functionality that is used in all products in the system family.

- Applications: Implementing the specific logic for a certain GSN node.

In the newest version of GSN the only node type based on this architecture is SGSN.

3.4.3 Definition of Architecture

GSN definition of architecture [GSNSAD02]: Software architecture is concerned with decomposition of a system into components, their interrelationships, and the principles and guidelines governing the design and evolution over time. Explicit definition of the software architecture allows:

- Communication with stakeholders who set the requirements.
- Assessment of quality requirements.
- Clearly defined purpose and responsibility of components, which makes it possible to get an understandable and evolvable system.

System architecture is more about an internal viewpoint of the system, than about an external (outside) viewpoint of the system. The external viewpoint is covered by system requirements.

3.4.4 Architectural Views

Every aspect cannot be captured in one view of the system. Instead several views of the system is used, each focusing on one dimension or usage, i.e. logical structure of components in the logical view, distribution of logic on processing resources in the deployment view etc. These views must also relate to each other to be able to go from one view to another view.

There is also a need for describing the architecture on different abstraction levels. Different people in the organization, like project managers, designers and testers have different tasks and responsibilities, and because of that they need different entries into the system. The abstraction levels on which the system is described are analysis, design and implementation.

3.4.5 Application Platform and Framework

Since more than one application (node type) should be developed, Ericsson decided to develop an application framework, called Capella, to support the applications with a lot of tasks, i.e. distribution, start and supervision of application logic, node internal communication services and resource handling. The framework consists of both run-time components and design/implementation rules to be followed when using the application framework. All components within Capella were made generic, trying to provide an application framework to be used by any packet handling application.

On the application is built on top of the application framework and platform. Some functionality on the application level is the same in SGSN and GGSN and because of that some components have been classified as “business specific” and grouped into an own package. The packages are called High Level Packages.

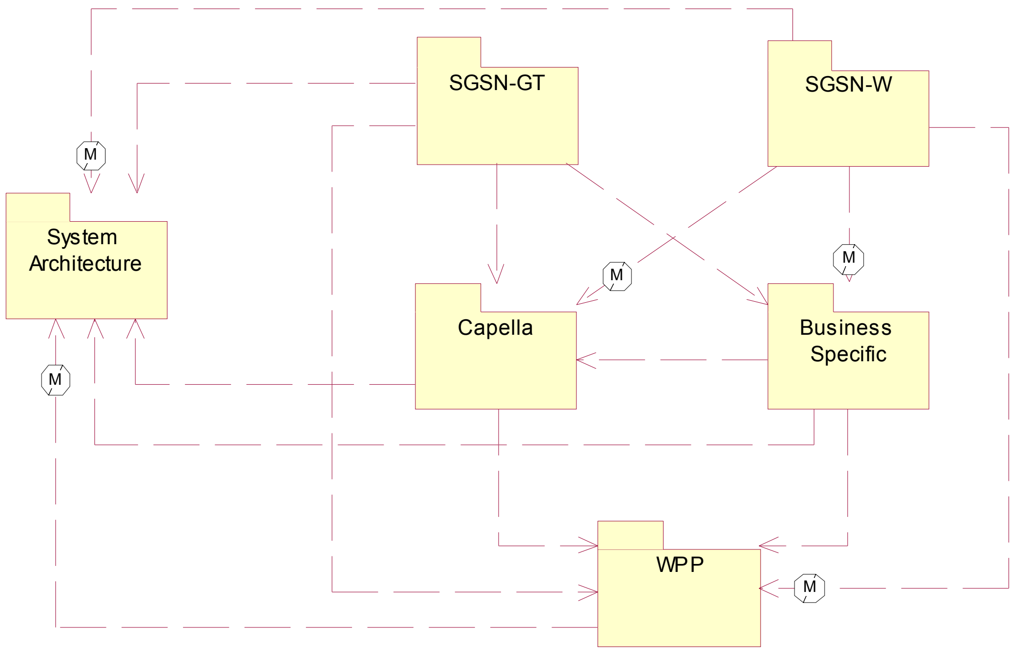


Figure 3-20: High level packages and their dependencies

The dashed arrows indicate that one high level package is dependent on another. Figure 3-20 can also be viewed as a layered architecture as shown in figure 3-21.

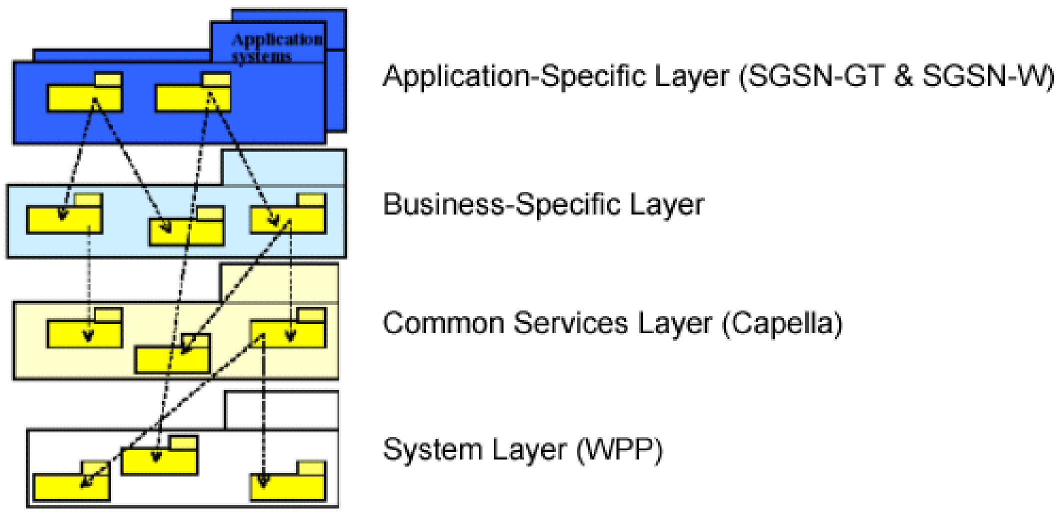

Application systems
Application-Specific Layer (SGSN-GT & SGSN-W)
Business-Specific Layer
Common Services Layer (Capella)
System Layer (WPP)

Figure 3-21: Layered architecture

3.5 GSN UML Model

[GSNMOD02] Ericsson has decided to use the Rational Rose modeling tool, hence all modeling activities are carried out in this tool as far as possible. The SGSN system is modeled using object oriented modeling techniques, and the models are described in the Unified Modeling Language (UML). This chapter describes the Logical View and the Component View of the model, focusing on the modeling of components.

3.5.1 Logical View

This view describes the logical structure of the system architecture, i.e. how the system has been divided into components and how those relate to each other.

3.5.1.1 Analysis Model

In order to support a high level reasoning about the system there is need for a high level decomposition of the system logic. The analysis model is aimed to serve that purpose and consists of analysis classes, which represents important entities, mainly originating from the problem domain but at the same time reflecting how the system realized.

3.5.1.2 Design Model

The design model describes the system in terms of design elements (components) with defined responsibilities and interfaces. The main purpose with the design model is to specify such components and to show the interactions between those in different situations.

Design Elements

Ericsson has defined the following terms which also are referred to as *components*:

- **Subsystem:** The highest level of encapsulation used. A subsystem has formally defined interfaces and is a collection of function blocks.
- **Block:** Contains formally defined interfaces and is a collection of lower level (software) units. A block often implements the functionality represented by one or more analysis classes in the analysis model.
- **Unit:** A collection of (software) modules, i.e. classes/objects. Two units within the same block may communicate without going through an interface.
- **Module:** Corresponds to a source code file (Erlang or C). Except for the interface modules generated from the interfaces on subsystem and block level, source code only exists on the unit level.

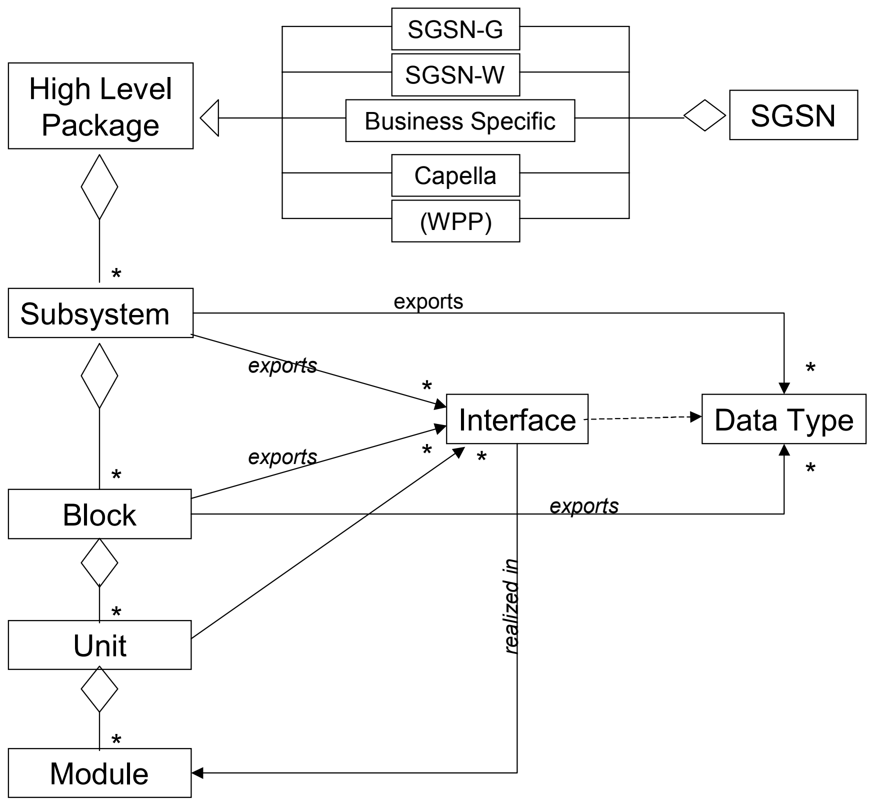


Figure 3-22: Different types of design elements that are used in the design model

Use of Packages

Subsystems, blocks and units are defined as packages in Rose in order to show aggregation (see figure 3-22). To give an overview of how certain subsystems, blocks and units relate or depend on each other, dependency arrows are used between packages to display dependencies. Figure 3-23 is an example of a unit dependency diagram.

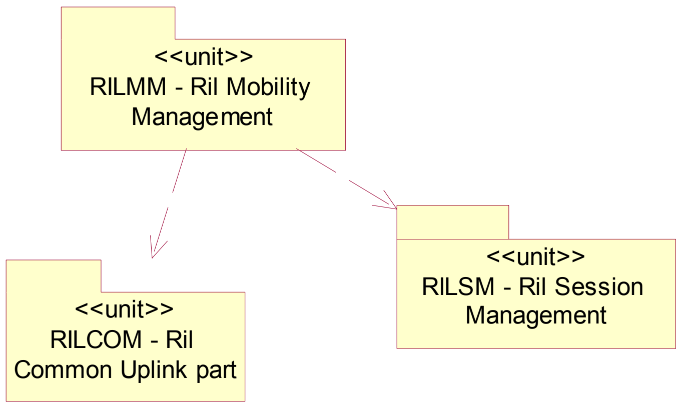

<<unit>>
RILMM - Ril Mobility Management
<<unit>>
RILCOM - Ril Common Uplink part
<<unit>>
RILSM - Ril Session Management

Figure 3-23: Example of a unit dependency diagram

Use of Classes

Subsystems, blocks and units are defined as classes in Rose. Each type has a proxy class (<<subsystem proxy>>, <<block proxy>> and <<unit proxy>>) that represents the corresponding packages. The reason for this is that Rose does not allow associations between packages and classes and you can not use packages in sequence and collaboration diagrams. The class-representation of the packages is only used in diagrams where realizations and inheritance are shown. The interface class is shown as an icon (the so called "lollipop") and a realize association to the corresponding subsystem, block or unit proxy class is used. This will enhance the readability of the model. Operation names defined in the interface will be available in the proxy class. Figure 3-24 shows an example of a subsystem class diagram.

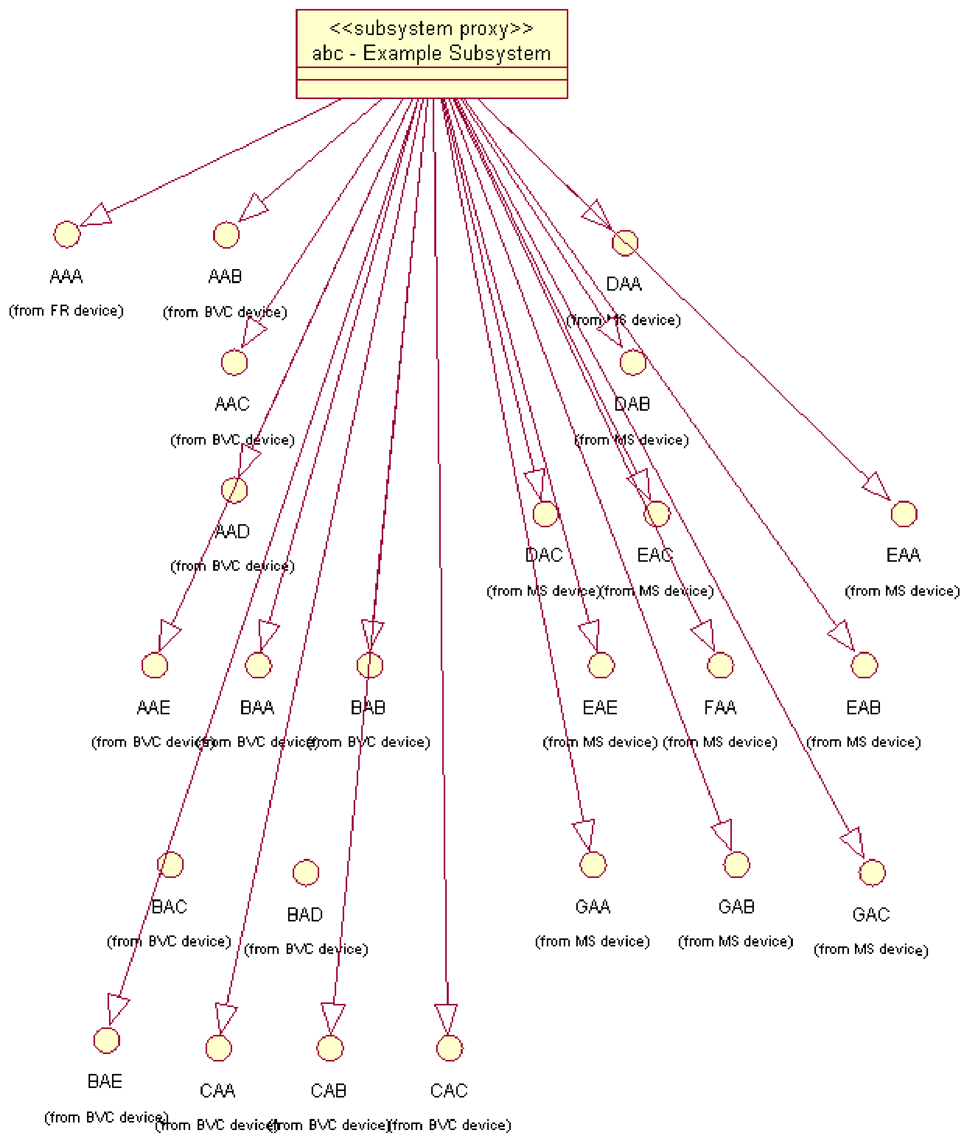


Figure 3-24: Subsystem class diagram example

Use of Interfaces

In the GSN model there are both interfaces as well as abstract interfaces.
Interfaces that are going to be exported out of a subsystem are still defined on block level and then inherited up to the subsystem level. An abstract interface is not instantiated, but inherited by another interface that is realized in another block or subsystem. The mechanism of abstract interface is usually used to design a callback interface.

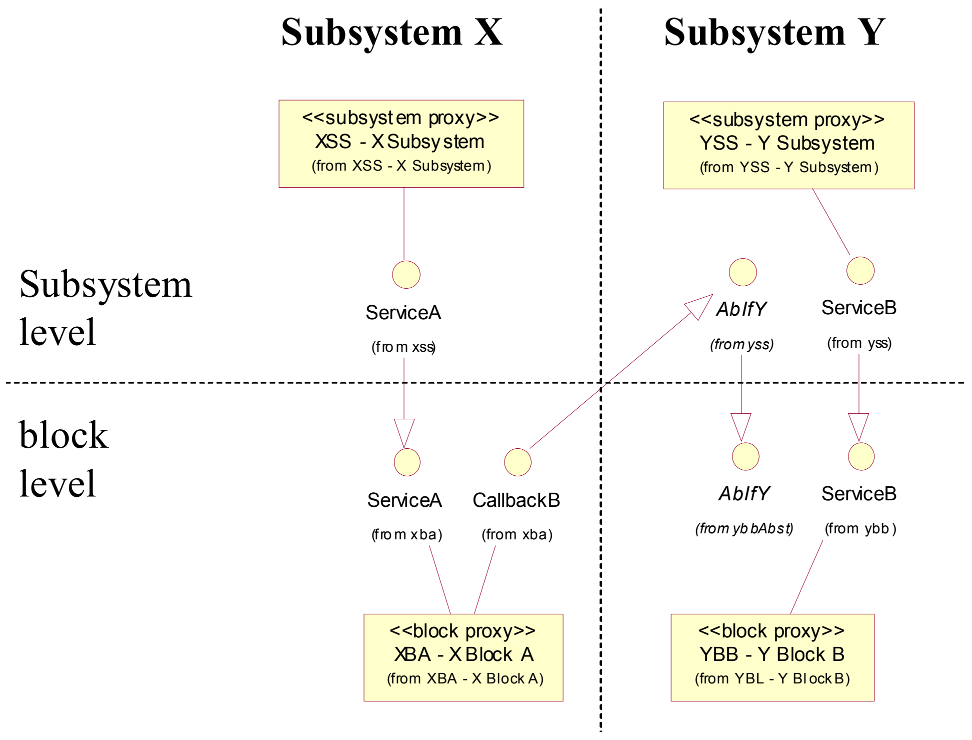

Subsystem X
Subsystem Y
<<subsystem proxy>> XSS - X Subsystem (from XSS - X Subsystem)
<<subsystem proxy>> YSS - Y Subsystem (from YSS - Y Subsystem)
Subsystem level
ServiceA (from xss)
AbIfY (from yss)
ServiceB (from yss)
block level
ServiceA (from xba)
CallbackB (from xba)
AbIfY (from ybbAbst)
ServiceB (from ybb)
<<block proxy>> XBA - X Block A (from XBA - X Block A)
<<block proxy>> YBB - Y Block B (from YBL - Y Block B)

Figure 3-25: Interface example

In figure 3-25 there are two subsystems; XSS and YSS.

- XSS has a block XBA that provides an interface called ServiceA to a service implemented by XBA. ServiceA shall be exported out of XSS, so it is inherited up to the subsystem level.
- For some of the services accessed through ServiceB, a callback interface is needed. The service provider YSS does not know who uses this service but defines how the callback interface must look like in the abstract interface AbIfY*.* AbIfY is first defined on block level in YBB, then inherited to YSS and exported.
- The block XBA is a service user to ServiceB and inherits the abstract subsystem interface AbIfY into its interface CallbackB. CallbackB is realized by the block XBA.
- The interfaces ServiceA and ServiceB may be used when blocks in the XSS subsystem are communicating via interfaces.

3.5.2 Component View

In the current version of UML and Rational Rose it is not possible to generate IDL code from the model in the logical view. It is therefore necessary to specify UML components for each subsystem, block and unit in the component view, and remember to use the same names in the logical view and component view. Whenever a design change is made in the logical view the designer has to make the equivalent change in the component view in order to keep the model in sync. The component view does not add any new information.

In the design model in the component view there is one package for each subsystem. Each of these packages contains a component which is equivalent to the corresponding subsystem in the logical view. The component has associations to all interfaces it provides as shown in figure 3-26. Additionally the subsystem package contains packages for each block it contains.

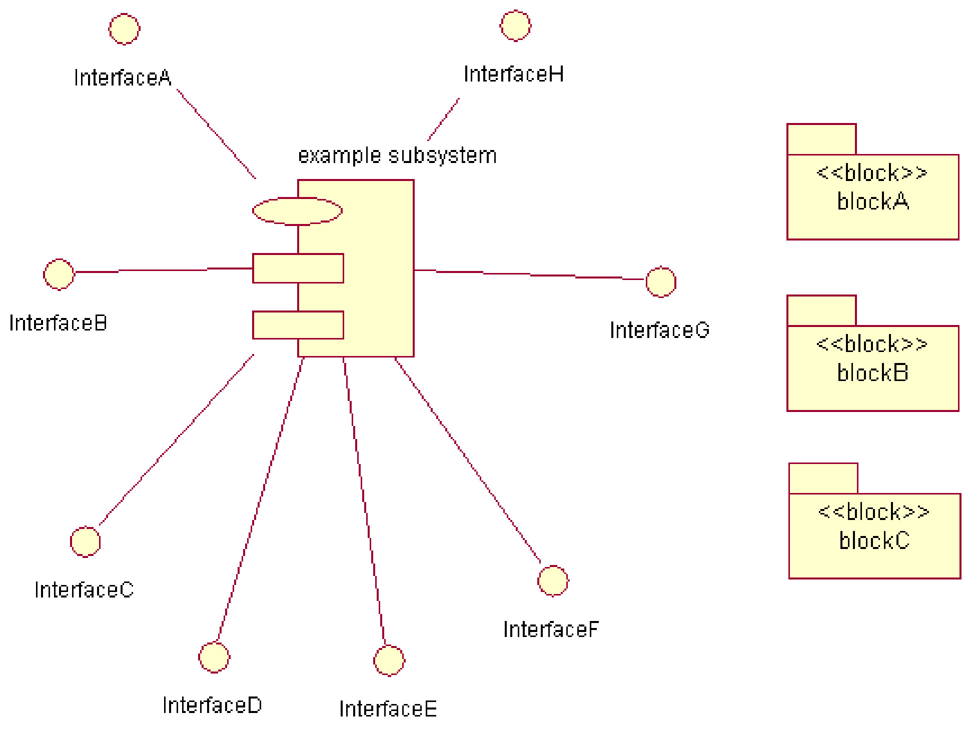


Figure 3-26: Component diagram example

3.5.3 Summary

The specification of components (subsystems, blocks, units and modules) is done by creating packages and stereotype classes in the logical view of the design model. The packages are used for aggregation of components and visualization of dependencies. Corresponding stereotype classes are used to realize interfaces and make the components available in sequence and collaboration diagrams. In order to generate IDL code the designer has to create UML components in the component view for all subsystems, blocks, units and modules defined in the logical view, and also associate all the used interfaces.

Recently the project has allowed that interfaces are only defined in the logical view and the IDL code will be written by hand. This means that Ericsson doesn't need to have two parallel views. However this means that some inconsistence may happen: Designers may change an IDL file without updating the model.

3.6 Evaluation of Current Practice

In this chapter we evaluate the current practice of both GSN RUP and the GSN UML model.

3.6.1 GSN RUP

In our study we found that GSN RUP does not have an activity that forces the designer to identify already existing components, also known as developing *with* reuse. Also GSN RUP lack a process of defining to what extent a component is developed *for* reuse [Naalsund01]. The same observations were done in [Naalsund01]. [Naalsund01] proposed new reuse activities to the GSN RUP adaptation, but these activities have not yet been included.

We also noticed that GSN RUP has few or no guidelines/activities on framework engineering. Framework engineering in GSN RUP is done by relying on experience since the framework responsible has few or no guidelines to rely on (as described in chapter 3.3.4 and 3.3.5).

Summary of problems:

1. No support for Product Line Engineering (no separate overall software framework activity). See chapter 4.2 for improvements suggestions.
2. Still no specific activities for developing components *with* and *for* reuse. See chapter 4.3 (with and for reuse) for improvements suggestions.

3.6.2 GSN UML Model

In our study of the GSN UML model and the use of Rational Rose (see chapter 3.5) we found some major shortcomings regarding the current modeling practice.

We think that the way Ericsson uses UML / Rational Rose to specify components and generate IDL code is inefficient and not very elegant. It produces duplication of work because the designer has to maintain two different views of the model. The readability of the model is also reduced due to the use of both packages and stereotypes classes in the logical and component view. Ericsson has expressed a need for an improvement in this area.

Summary of problems:

- Inadequate support for component specification.
- Duplication of design and maintenance work.
- Reduced readability of model due to use of both packages and stereotype classes.

See chapter 4.4 for improvement suggestions.

4 Improvement Suggestions

4.1 Introduction

This chapter contains our improvement suggestions for GSN RUP. We use the-state-of-the-art processes and standards, explored in chapter 2, to suggest improvements to the problems identified in chapter 3.

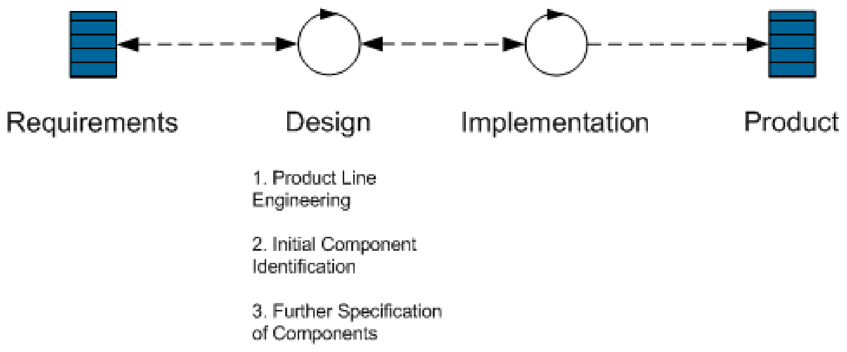

Requirements
Design
Implementation
Product
1. Product Line Engineering
2. Initial Component Identification
3. Further Specification of Components

Figure 4-1: Overview of the problem areas identified in GSN

4.2 Product Line Engineering in GSN RUP

The study of Ericsson's current practice (chapter 3.3.5) shows that GSN RUP lacks specific activities for framework engineering.

KobrA could be used to support a product line approach within the overall project management umbrella of the GSN RUP. Framework engineering is done in a more ad-hoc manner and the framework responsible has few or no guidelines to rely on. We feel that the KobrA framework engineering activities (see 2.1.3.3) in combination with GSN RUP application engineering activities could be combined to form a separate framework engineering activity in GSN RUP.

Advantages:

- Improved guidelines on the framework engineering process in GSN RUP.
    - This will provide more standardized design.

Disadvantages:

- Costly to implement.

4.3 Initial Component Identification in GSN RUP

The study of Ericsson's current practice (chapter 3.5) shows that GSN RUP lacks activities for identification of reusable components and evaluation of these. This work is now experience-based.

Our studies of RAS have discovered that RAS provides a common way for describing components so that they easily can be identified from a component repository. Rational Software has set forward a set of guidelines for integration of RAS in RUP. Appendix A contains a summary of the integration proposal collected from [RAS02]. From these circumstances it seams like RAS would fulfill the design *with* and *for* reuse needs of the GSN project. With the suggested new RUP roles and workflow details it should be a reachable goal to integrate RAS in RUP. At least in the theory, since changing the current practice will always be time-consuming and require expert insight in the existing RUP process.

We feel that using the variant functionality exploring described and used in KobrA (as well as in other methods) can help to identify components (see chapter 2.1.3.4). The analyzing and encapsulating of the variabilities that characterize a product line can provide valuable insight into how best to consolidate functionality into good, reusable building blocks for the domain in question. Factoring out and encapsulating variabilities is in fact a way of identifying components. This could help in the process of evaluating what components that should be added to the framework. We therefore recommend modifying GSN RUP to capture this activity. Such a modification would probably not require large effort.

COMET (see chapter 2.1.4) is prescriptive about both the management aspects and the technical aspects of a software project. It is therefore not easily integrable with RUP. In addition COMET does not have a full life cycle methodology. It will not provide any improved processes to GSN RUP and we therefore do not recommend COMET integration with GSN RUP.

Advantages:

- RAS provides GSN RUP with activities for searching and identification of components.
- RAS is easily adaptable with GSN RUP.
- Variant functionality exploring can help component identification.
- Variant functionality exploring is cheap to integrate.

Disadvantages:

- Time consuming maintaining of the component repository.
- Most component repositories have little or no use [Poulin95].

4.4 Further Component Specification in the GSN Model

Our study of the current component modeling practice at Ericsson using UML 1.x (chapter 3.5) expresses a need for an improvement on component specification. The problems originate from the fact that components are not logical entities in the logical view of UML 1.x and RUP. This shortcoming forces Ericsson to work around this problem by modeling components using packages and stereotype classes. In order to be able to generate IDL code they are also forced to specify the same components in the component view of Rose, creating duplication of design and maintenance work.

After studying the new CCM standard (chapter 2.2.2) for extending CORBA with components we think that CCM might be used to improve component specification in GSN. Because the GSN system is already based on CORBA this favors the integration of CCM versus other component standards. The port mechanisms of CCM enhance component reusability and exchangeability. The new UML 2.0 standard (reviewed in chapter 2.2.1) could be used to model a CCM based system. The new component constructs of UML 2.0 matches the ones in CCM (i.e. the use of ports for specifying provided and required interfaces). The use of CCM and UML 2.0 would allow specification of components in the logical view of the model. We think that this approach would enhance the support for component-based development and increase the readability of the model. It would also lead to more efficient design work.

However CCM is a large and complex specification. If Ericsson starts using CCM this would require training of employees, purchase of new tools and updates of artifacts such as the software architecture description and modeling guidelines. Hence a conversion of the GSN architecture to CCM is an expensive in operation. CCM is also a new standard, hence little practical experience and limited tools are available. UML 2.0 is not yet a finished standard, so further evaluation must be postponed until the standard is finished.

We recommend Ericsson to further investigate the possibility of using CCM and UML 2.0. A risk assessment and cost vs. benefit analysis would be of high importance. Also a more in-depth technical study of how these new standards may be integrated.

The component modeling aspects of KobrA (see chapter 2.1.3) could be applied within the GSN RUP framework to provide a more rigorous and component oriented approach for using the UML. This is however not plausible when the proposed UML 2.0 standard takes over from the current version of UML. UML 2.0 will cover the needs Ericsson has on component specification.

Proposed improvement:

- Use CCM to specify components in CORBA
- Model CCM system with UML 2.0

Advantages:

- Improved component specification
- Use of ports increase component reusability
- More efficient design work
- Increased readability of model

Disadvantages:

- Expensive integration and high risk
- Large and complex specifications
- New standard, hence little practical experience is available
- Limited tool support

4.5 Summary

In this chapter we have presented improvement suggestions to the GSN RUP process. To retain high quality in the software development process after changing it and to check whether the suggested changes are followed up in practice, it would be necessary to measure the changed performance of the process. There are many different standards and methods from the field of Software Process Improvements (SPI) that may support this task. One example of a framework for process improvement is SPIQ (Software Process Improvement for Better Quality), refer to [SPIQ00]. Goal Question Metric (GQM) could be used to validate the proposed changes.

There is always an evaluation of cost vs. benefit when changing already existing processes and standards. This is consideration that Ericsson must make. The improvements could be applied incrementally, thus distribute cost and risk over time.

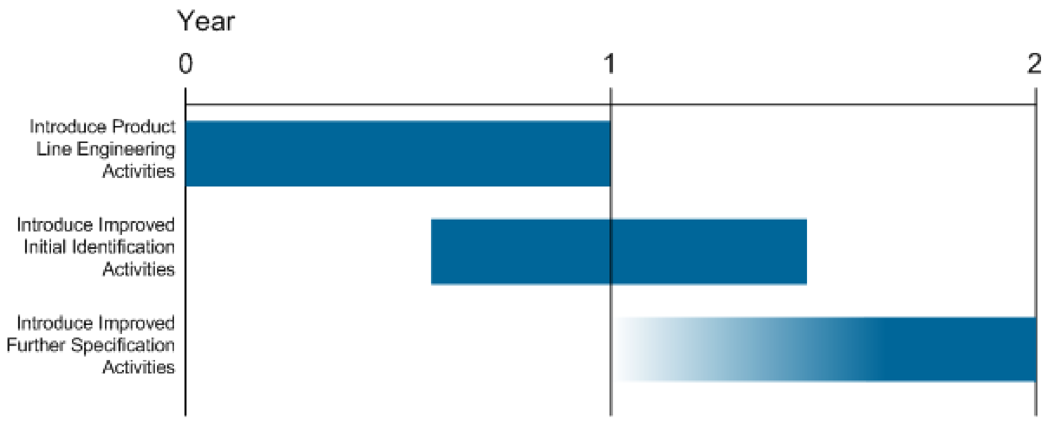


Figure 4-2: Proposed timeline for introducing our improvements at Ericsson

In figure 4-2 present a timeline for introducing our improvements at Ericsson. This is our qualified guessing, because we have no foundation to base our estimation on. This will depend on the resources allocated for the improvement adaptation. The time aspect is of less importance, it is the sequence that is the main issue.

The timing of the introduction of a further improved component specification activity is uncertain because the UML 2.0 standard is not yet finished.

"Nothing is more difficult than to introduce a new order. Because the innovator has for enemies all those who have done well under the old conditions and lukewarm defenders in those who may do well under the new."

- Niccolo Machiavelli, 1513A.D.

5 Conclusion and Further Work

After studying and comparing the state-of-the-art of development processes and standards, and the current practice at Ericsson, we have discovered that the there is an improvement potential for the GSN RUP process, especially on component identification. We also believe that the current modeling practice regarding component specification can be improved.

We suggest that Ericsson could use experiences from KobrA and modify their GSN RUP process, especially include a product line engineering approach. We recommend KobrA's variant functionality exploring can be used to identify components in the development process.

Exploring Rational's guidelines for integration of RAS in RUP, we found that RAS could be easily integrated into GSN RUP and we recommend Ericsson to consider this possibility more closely.

Our study of the new UML 2.0 standard concludes that Ericsson may investigate further the possibility to start using UML 2.0 as soon as the standard is completed. We found that UML 2.0 will suit Ericsson's need for improved component specification. We also recommend Ericsson to review the possibility of using CCM. UML 2.0 could then be used to model the CCM system. But such a decision would allocate large amounts of resources, and thus a cost vs. benefit analysis and risk assessment would be of high importance.

For further work, we propose the following:

- Explore the possibility of using experiences from KobrA's component identification process and product line approach for improving GSN RUP. We recommend developing this solution further, making a plan for changing RUP and a plan for evaluating the changes, using methods from SPI.
- We have presented Rational's suggested integration of RAS in RUP. This integration should be further developed, tested and verified empirically.
- Further explore the possibility of integrating UML 2.0 and CCM into the modeling practice at Ericsson. This would include a more detailed feasibility study and an analysis of suggested modeling practice with UML 2.0 and CCM. It would be possible to test UML 2.0 / CCM in a pilot project or a constructed case study. A survey could be conducted with the developers at Ericsson.

6 References and Abbreviations

[Atkinson00]	C. Atkinson, J. Bayer, O. Laitenberger and J. Zettel: “Component-Based Software Engineering: The KobrA Approach”, Fraunhofer IESE, 2000
[Atkinson01]	C. Atkinson, J. Bayer and D. Muthig: “Component-Based Product Line Engineering - The KobrA Project”, Fraunhofer IESE, 2001
[Atkinson02]	C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst and J. Zettel: “Component-Based Product Line Engineering with UML”, Fraunhofer IESE, 2002
[Boehm99]	B. Boehm and C. Abts: “COTS Integration: Plug and Pray?”, IEEE Computer, January 1999, pp. 135-138
[Boehm02]	B. Boehms: “Get Ready For The Agile Methods, With Care”. Computer 35(1): 64-69
[Boertien01]	N. Boertien, M.W.A. Steen, H. Jonkers: “Evaluation of Component-based Development methods”, Telematica Instituut, 2001
[Carney97]	D.J. Carney and P.A. Oberndorf: “The Commandments of COTS: Still in Search of the Promised Land”, Crosstalk, May 1997, Vol. 10, No. 5, pp 25-30
[Cheesman01]	J. Cheesman and J. Daniels: “UML Components”, Addison-Wesley, 2001
[Conradi01]	P. Mohagheghi and R. Conradi, "Experiences with certification of reusable components in the GSN project in Ericsson, Norway", 4th ICSE Workshop on Component-Based Software Engineering, Toronto, Canada, May 14-15, 2001
[Ericsson02]	“Ericsson in Brief”, http://www.ericsson.com/about/compfacts, 18.10.02
[Fowler00]	M. Fowler and K. Scott: “UML Distilled”, Second Ed., Addison-Wesley, 2000
[Gamma95]	E. Gamma, R. Johnson, R. Helm and J. Vlissides: “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995
[Garlan95]	D. Garlan, R. Allen and J. Ockerblom: “Why reuse is so hard”, IEEE Software 12(6):17-28, 1995
[GSNSAD02]	GSN Software Architecture Description (SAD), Rev. A, 08.08.02, (Ericsson Internal)
[GSNRUP02]	GSN RUP Adaptation, release R4A, Ericsson Intranet (not public), 2002
[Kobryn00]	Cris Kobryn and Thomas Weigert: “OMG UML 2.0 RFPs” (slides), OMG, 2000

[Kobryn02]	C. Kobryn: "Will UML 2.0 Be Agile or Awkward?", Communications of the ACM, pages 107-110, Jan. 2002 / Vol. 45, No. 1
[Kruchten00]	P. Kruchten: "The Rational Unified Process, An Introduction", Addison Wesley, 2000
[Larsen02]	G. Larsen: "The Asset Browser", 2002, see http://www.rational.net
[Morisio00]	M. Morisio, M. Ezran and C. Tully: "Investigating and Improving a COTS-Based Software Development Process", Proc. 22nd International Conference on Software Engineering (ICSE'2000), pages 31-40, ACM Order No. 592000
[Naalsund01]	E. Naalsund, O. Walseth: "Component-based development. Models for COTS / Software Reuse", student project, NTNU, 2001
[Naalsund02]	E. Naalsund and O. Walseth: "Decision-Making in Component-Based Development", Diploma Thesis NTNU, 2002
[OMG02-1]	UML 2.0 Superstructure RFP, version 2 beta R1 (draft), OMG, 2002
[OMG02-2]	"CORBA Components, ver. 3.0", CCM Specification, OMG, 2002
[Poulin95]	J. S. Poulin: "Populating Software Repositories: Incentives and Domain-Specific Software", Journal of System and Software, p. 187-199, 1995
[RAS02]	RAS Specification, see http://www.rational.net (09.04.2002)
[RUP02]	RUP 2002 web site, see http://www.rational.com/products/rup/index.jsp
[Sjøberg00]	D. Sjøberg (UiO) and R. Conradi (NTNU): "Incremental and component-based software development (INCO)", 2000
[Solberg97]	A. Solberg and A. Berre: "Component Based Methodology Handbook", SINTEF, 1997
[SPIQ00]	SPIQ (Software Process Improvement for Better Quality), Metodehåndbok, 2000 [http://www.geomatikk.no/spiq/]
[Stoj01]	Z. Stojanovic, A. Dahanayake, H. Sol: "A Methodology Framework for Component-Based System Development Support", Faculty of Information Technology and Systems, Delft University of Technology NL, 2001
[Wang00]	N. Wang, D. Schmidt and C. O'Ryan: "Overview of the CORBA Component Model", 2000

Abbreviations

A&D	Analysis & Design (see 3.3 GSN RUP)
ARS	Application Requirements Specification (see 3.3 GSN RUP)
CBSE	Component Based Software Engineering (see 2.1.1)
CCM	CORBA Component Model (see 2.2.2)
CDL	Component Definition Language (see 2.2.2 CCM)
COM	Component Object Model (see 2.1.3 KobrA)
COMET	Component Based Methodology (see 2.1.4)
CORBA	Common Object Request Broker Architecture (see 2.2.2 CCM)
COTS	Commercial Off The Shelf (see 2.1.1 CBSE, 2.1.3 KobrA and 3.3 GSN RUP)
EJB	Enterprise Java Beans (see 2.1.3 KobrA and 2.2.1 UML)
FIS	Feature Impact Study (see 3.3 GSN RUP)
J2EE	Java 2 Enterprise Edition (see 2.1.3 KobrA and 2.2.1 UML)
GQM	Goal Question Metrics (see chapter 4.5)
GSN	GPRS Support Node (see chapter 3)
IDL	Interface Definition Language (see 2.2.2 CCM)
OMG	Object Management Group (see 2.1.4 COMET, 2.2.1 UML and 2.2.2 CCM)
RAS	Reusable Asset Specification (see 2.2.3)
RFI	Request for Information (see 2.2.1 UML)
RFP	Request for Proposal (see 2.2.1 UML)
RUP	Rational Unified Process (see 2.1.2)
SAD	Software Architecture Description (see 3.3 GSN RUP)
SPI	Software Process Improvement (see chapter 4.5)
UDDI	Universal Description, Discovery and Integration (see 2.2.3 RAS)
UML	Unified Modeling Language (see 2.2.1)
WSDL	Web Service Definition Language (see 2.2.3 RAS)
XDE	eXtended Development Environment (Rational Tool, see Appendix A)
XML	eXtensible Markup Language (see 2.2.3 RAS)

Appendix A: Details on RAS Integration in RUP

New RUP Roles

When integrating the Reusable Asset Specification into the Rational Unified Process, [RAS02] introduces five new RUP roles. These roles are:

- Asset Consumer: This is could be a developer, an architect, a designer which installs the asset into their environment and customizes it for their context.
- Asset Evaluator: Every person which browses asset and artifact documentation is an Asset Evaluator.
- Asset Harvester: This is typically an architect, designer, or team who makes asset size, type, and boundary decisions and prepares artifacts to be packaged as an asset.
- Asset Packager: The duty of the Asset Packager is to collect the artifacts and organize them for Asset Evaluators and Asset Consumers.
- Asset Librarian: The Asset Librarian makes the asset visible and tracks relevant metrics.

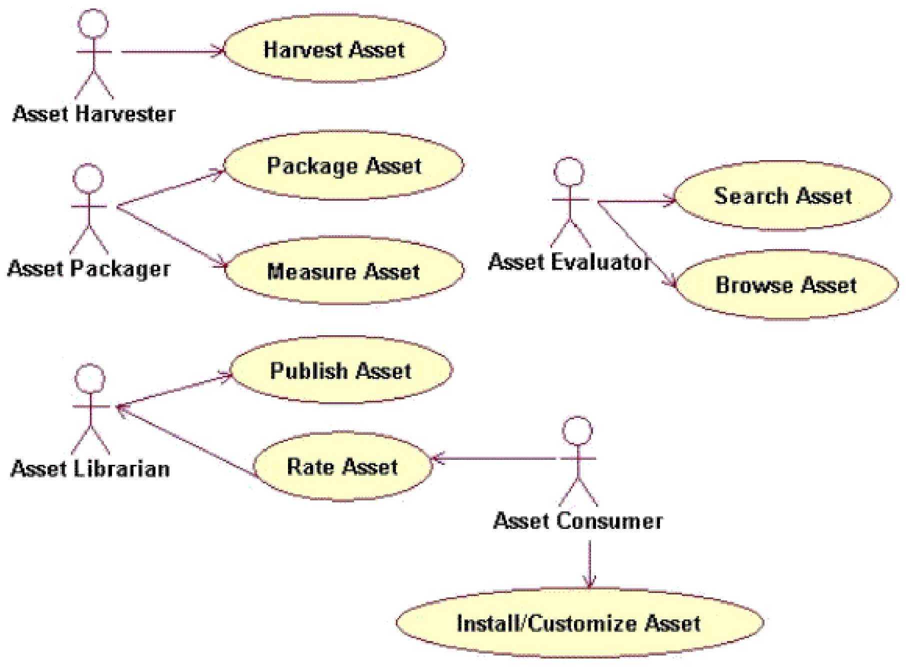


Figure: New actors and the use cases with which they interact

New RUP Workflows Details

Rational here introduce eight new workflow details. These are grouped into production, management and consumption.

Asset Production

These workflows describe the activities associated with the definition, creation and production of assets.

- Harvest Asset: Define the boundaries and scope of asset and prepare the artifacts to be reused.
- Package Asset: Document and organize artifacts for the purpose of searching, browsing and usage.

Asset Management

These workflows describe the activities of the management of assets-based development.

- Publish Asset: Transition the asset through acceptance states and then make it visible to the appropriate audience.
- Measure Asset: Capture asset metrics.
- Rate Asset: Collect and report asset usage reviews.

Asset Consumption

These workflows describe the activities associated with the evaluation and usage of assets.

- Search Asset: Identify and submit search criteria to evaluate candidate assets.
- Browse Asset: Review and evaluate asset artifacts.
- Install/Customize Asset: Insert asset into your workspace and extend where necessary.

The most important are Harvest Asset and Package Asset. We will describe these in more detail.

The Harvest Asset Workflow

The Harvest Asset workflow defines the boundaries and scopes of the asset and prepares the artifacts to be reused. . The Asset Harvester should consider the artifacts reuse potential and their relevance to solving repeating problems. Here the reusable artifacts are identified, the generic elements introduced and the non-reusable context is removed.

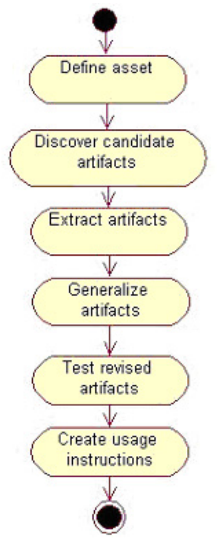


Figure: Activity diagram of the Harvest Asset workflow

Workflow Description

This workflow begins when an Asset Harvester determines that the conditions for creating and using an asset have been met regarding a group of artifacts. This workflow assumes that the necessary analysis has been performed to determine that an asset should be created. This workflow focuses on the steps for determining the scope and preparing the artifacts of the asset. The main role in this workflow is the Asset Harvester.

Mapping the Workflow to RUP

Since assets may contain artifacts from all aspects of the software development lifecycle, this workflow may be performed at the end of Inception, Elaboration, Construction, and Transition iterations.

Artifacts Delivered

This workflow typically produces the following artifacts:

1. Asset documentation describing

   a. The definition of the asset and its type.
   b. The initial classification of the asset.
   c. The variability points and the possible items they may be bound with.
   d. The initial usage instructions.
   e. The identification of related assets.

2. An XDE project with the generalized artifacts.

The Package Asset Workflow

The Package Asset workflow documents and organizes artifacts for the purpose of searching, browsing and usage. This workflow is may be the most important because it has a direct impact on the asset's comprehension and reusability.

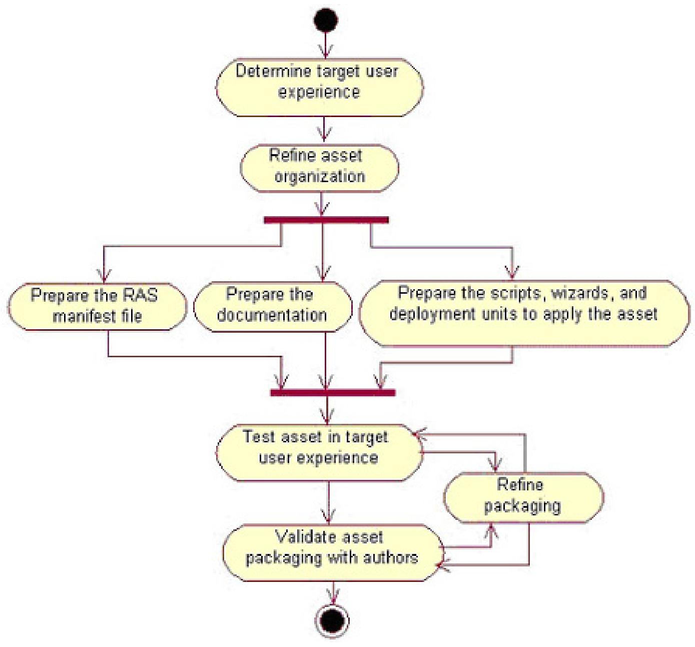

Determine target user experience
Refine asset organization
Prepare the RAS manifest file
Prepare the documentation
Prepare the scripts, wizards, and deployment units to apply the asset
Test asset in target user experience
Refine packaging
Validate asset packaging with authors

Figure: Activity diagram of the Asset Packager workflow

This workflow begins when an asset has been defined and scoped and its artifacts are prepared for reuse. The Asset Packager focuses on organizing the asset into the RAS format with the emphasis on supporting asset evaluation and usage. Typically the Asset Harvester delivered some artifacts to the Asset Packager. The deliverables from this workflow will be published in some format for Asset Evaluators and Consumers to access. The Asset Packager should have writing skills and should focus on organizing the asset for a particular audience accessed through a specific channel.

Mapping to RUP Phases

This workflow should be performed after the Harvest Asset workflow. Therefore, this workflow may be performed at the end of Inception, Elaboration, Construction, and Transition iterations.

Artifacts Delivered

This workflow typically produces the following:

1. External documentation, for example documentation for asset browser software.
2. The updated documentation and other artifacts.
3. The .ras file(s).