Master Thesis
Software Engineering
Thesis no: MSE-2005-06
June 2005

Reverse Architecting
– A case study of workbench approach

Gang Sun

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Contact Information:
Author:
Gang Sun
Address: Ölandsgatan 1-3, S-371 23, Karlskrona
E-mail:gasa03@student.bth.se

External advisors:
PerOlof Bengtsson, Olle Blomgren
Ericsson AB
Address: Ölandsgatan 1-3, S-371 23, Karlskrona
Phone: +46 455 395000

University advisor:
Olle Lindeberg
School of Engineering

ABSTRACT

Architecture is a key factor to the success of the software product, but at least the state of art is not the state of practice. The connection between the completed system and the architecture has often been damaged after the original implementation. Reverse architecting is a method to recover the software architecture by analyzing the artifact. Since reverse architecting is not mature yet, the paper studies methods, tools, and major issues which hinder the person performing reverse architecting all together from knowing whether the reverse architecting method is repeatable and possible to apply on the different releases of the same product.

Keywords: Reverse architecting method, Artifact, Design, Rational XDE, Round-trip engineering

ACKNOWLEDGEMENTS

I would like to thank all people that have contributed to the thesis especially …

… the node architects and designers at Ericsson AB who participated in the interviews

… my advisors Olle Lindeberg at BTH and PerOlof Bengtsson and Olle Blomgren at Ericsson AB

… Ericsson AB for putting personnel as well as resource for the thesis

CONTENTS

1 INTRODUCTION

One Ericsson [24] product is selected for the case study. It is a real-time distributed software product with about thousands of lines of C++ code.

We have investigated all artifacts related to the product in the case study, for example the code, the UML diagrams and all other documentation. The investigation shows that the product has incomplete documentation of the software architecture that hinders the continuous development for the next release of the product, and prevents the organization from efficiently catching the rapidly changing market.

Own way to improve the situation is to use reveres architecting to reconstruct the software architecture from the existing code. This is not easy to do and we have to conclude that reverse architecting technology is not mature. There are many versions that do not work well.

1.1 Objective

The aim of this master thesis work is to find 1) a method to extract the concrete architecture from the source code and the available documentation to make the project close to the round-trip engineering and 2) evaluate how Rational XDE application do the reverse architecture. Reverse architecting is a part of normal maintenance - generate the architecture description from the implementation. Then the architect has the possibility to use the outcome to optimize the architecture and improve the implementation. The architecture of the existing product is an input for the next release in the future.

1.2 Chapter Outline

The paper starts with defining the terminology in chapter 2, and the reverse architecting approach is described in chapter 3 and the analysis of the method is in chapter 4. A case study by means of this method and a commercial tool and the reverse architect views are introduced in chapter 5 and chapter 6 discusses the conclusion. The appendix lists the supplementary information including the answers from interviews, and the adaptations of the tools.

2 TERMS

Here is an overview of some terms, each term is defined according to the usage in the paper. Component, node and system are 3 levels from bottom to top in a hierarchical description of a system. They are part of the realization of the architecture. Unified Modeling Language (UML) provides a standard and Architecture Description Languages (ADL) use this standard to describe the architecture in a formal way. Then in the maintenance phase Rational eXtended Development Environment (Rational XDE) reverses the architecture, and the architect analyzes the result and improves the implemented architecture. Round-trip Engineering is a goal of the reverse architecting.

2.1 Basic Concepts

Concept	**Definition**
Component	A component is a unit of composition with interfaces and quality attributes [1].
Node	A node consists of a set of components and their connections.
Product	A product in this paper is a part of a system. This means that a product is a node, which maintains what we discuss.
System	System consists of nodes and the interfaces between them.

Table 1. Basic Concepts

2.2 Architecture

[2] gives a description of the software architecture: ”the software architecture of a program or computing system is the structure of the system, which comprises software elements, externally visible properties of those elements and relationships amongst them. Software elements can be nodes, components and etc. whatever are defined in the system.”

A reasonable description sometimes is possibly more important than a thoroughly accurate definition of software architectures. It has to be mentioned that there is no single agreed-upon definition at all. The quotation above is one of the most often quoted. The common element in all definitions is the intuition that software architectures describe in different levels of detail, the inner workings of system which is much like the traditional architecture describes the structure of buildings, though the analogy is useful to help us to understand the concept and show cracks and differences if taken too far.

Some important and positive aspects of software architecture are [2]:

communication among stakeholders is improved when a reasonably accurate and high-level description of the system is available. It enables participants to effectively discuss the merits of the system and agree on a compromise which satisfies the requests of all the parties involved;

early design decisions are directly influenced by the architecture and greatly affect the subsequent development of the system.

abstraction reusability is another trait of software architectures that can be reused when another problem poses similar requirements or in a software product line;

better human organization can be achieved by matching the structure of the system to the structure of the development team;

Maintenance of existing systems can be made easier by referring to the software architectures;

It must be noted that architecture is a rather generic term and, depending on the objective with architecture it may be:

a conceptual architecture describes the system in terms of its major design elements and the relationships among them [17];

a concrete architecture is the as-built architecture and is the one we can recover using the tools. The paper will focus on reversing the concrete architecture and the result of the reverse architecting is a description of the concrete architecture;

a reference architecture is a sample from a well-known domain and is usually provided as a starting-point.

2.3 Architecture Description Languages

After the introduction of the definition and objective of the architecture, the next question is how software engineers describe the architecture accurately and reliably.

During the design phase, a very useful option is the so-called 4+1 View Model proposed [20] that tries to clarify the intended meaning of the traditionally used diagrams in which, for example, the nodes could be code sections, computers or functionalities while arcs could stand for dependencies or flows: by separating the descriptions into different views, each of them becomes more unambiguous. The model prescribes the following views:

logical view that deals with functional requirements;

process view that is concerned with nonfunctional requirements such as availability and performance;

deployment view that illustrates the work organization in the deployment group and constitutes the basis for the creation of product lines;

physical view that maps elements from other views to the underlying hardware;

scenarios or use-cases that serve to validate the other views and help discovering architectural elements.

It is more important to use formal notations to describe software architectures than informal ones. It is possible to verify the architecture properties defined in ADL. ADL comes in many flavors, ranging from the easy-to-use (at the expense of tool support) to strict formalisms (that allow for automated verification and code generation).

2.4 Unified Modeling Language

UML [26] is an industry-standard modeling language used for specifying, visualizing, constructing, and documenting the artifacts of a system.

One goal is to present a common modeling language that all developers can use. UML is a language whose vocabulary and rules focus on the conceptual and physical representation of a system.

The UML uses diagrams to represent different system views. The purpose of a diagram is to present a set of model elements, which are rendered as shapes and connectors.

As systems become complex, the importance of models increases. For example, a doghouse can be constructed without blueprints. However, as one progress to houses, and further to skyscrapers, the need for blueprints becomes pronounced. A model [25] helps to:

1. aid understanding of complex systems;
2. explore and compare design alternatives at a low cost;
3. form a foundation for implementation;
4. capture requirements precisely;
5. communicate decisions unambiguously.

UML is a standard that enables engineers to describe the architecture in a common way, and ADL uses this standard to describe the architecture.

2.5 Artifact

Artifact indicates anything (e.g. documentation, source code and etc) that can be used for the forward engineering and reverse engineering.

2.6 Design

Design is the process of how to come from the requirements to the implementation of the node. Design also documents the internal mechanism of the components in the artifact in the whole lifecycle of the design.

2.7 Round-trip Engineering

Round-trip engineering [4] is closely related to reverse engineering and it keeps architecture consistent. Assume that there is a reverse engineering procedure that is always able to give the architecture of a given product. Now assume that there is a procedure that will always generate the product from a given architecture. If the architecture of a given a product can be used to generate a product that is identical to the given product then this is a round-trip engineering system. Depending on the complexity of the given product, the round-trip engineering can be either automatically or human involved.

However, round-trip engineering is not only forward engineering and reverse engineering. Both of them involve transforming one or more artifacts into one or more other artifacts. This task is typically a single, isolated transformation, where any information in the target artifact is not considered, and typically a new artifact is created, possibly replacing the previous version if one exists. In some cases, forward and reverse engineering may be optimized so that only incremental transformation is performed, i.e. only the changed modules are transformed, rather than all artifacts, e.g., incremental compilation. For more information, see [19].

2.8 Rational eXtended Development Environment

Rational XDE [25] is a tool used for the case study in the paper. It combines the architecture design and C++ development into a tightly integrated environment. Rational XDE is a tool for designing, communicating, and documenting the architecture throughout the project lifecycle that seamlessly integrates with the development platform used to create applications and systems.

Rational XDE uses a variety of artifacts. Rational XDE facilitates a controlled iterative application-development process called round-trip engineering. The process models the application, analyzes and refines it as the understanding of its operation increases, then generates the code-element object declarations of a complete application based on that model. Rational XDE and the (Integrate Design Environment) IDE evolve the generated code, until the required functionality for the iteration is achieved.

3 REVERSE ARCHITECTING

3.1 Introduction

Reverse architecting uses artifacts to make the existing architecture explicit by extracting components and their relationships. There's no simple agreement on a single definition, for example:

[14] gives that reverse architecture recovers recurring designs and the rationales behind them;

[16] gives the main goal of reverse architecting - to retrieve high-level design structures, to reduce the effort, timescale and cost associated with modifying an existing system.

Since reverse architecting is part of reverse engineering, we use reverse engineering process to show the reverse architecting.

Reverse engineering is a process to analyze a system to identify its components and their interrelationships, creating representation of the system in another form or at a higher level of abstraction [15]. Reverse engineering is mainly used to ease the system maintenance and the future development, as well as to verify and update system documents.

Forward engineering is a traditional process that transforms high-level abstractions to a physical implementation [18].

Here are the definitions for some terms in Figure 1:

Requirements definition describes the goal, requirements, constraints and marginal conditions for a system;

Commercial requirements definition describes the requirements from the customer. It describes what the product must do;

Architecture definition defines the contents of the architecture. I.e. it includes the interfaces, connections and constrains among the components. The interface specifies the components behavior; the connection defines the communication between components; and the constraint restricts the behavior of the interface and the connection;

Implementation is the step that produces the final product.

Figure 1 shows an overview of the forward and reverse engineering in the software development cycle. The left and up arrow indicates reverse engineering, right and down arrow for forward engineering. The commercial requirement drives the architecture and forces changes in the architecture. The architecture dictates rules and guidelines to the requirements, design and implementation phase.

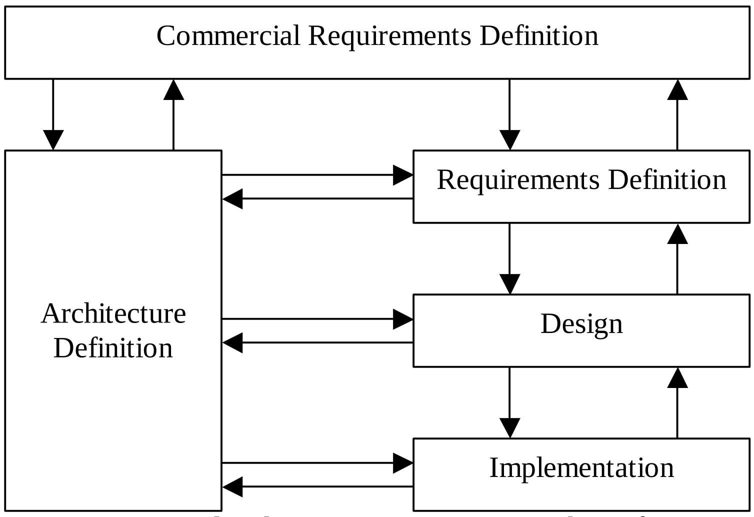

Figure 1. Forward and Reverse Engineering in the Software Development Cycle

Most people agree that there is a huge dimension during software maintenance and program understanding. [13] quotes the "Wall Street Journal" as source and reports that up to 80% of the maintenance effort is spent on trying to understand code and also [21] suggests a figure between 40% and 80%.

Many approaches in the past have developed and presented reverse architecting for:

evaluating the conformance of the as-built architecture to the documented architecture;

recovering the architecture description for systems that are poorly documented or for which documentation is not available;

analyzing and understanding the architecture of existing systems to enable modification of the architecture to satisfy new requirements and to eliminate existing software deficiencies.

The outcome of reverse architecting is the logic view and scenarios for the concrete architecture. This is one of the simple ways to show the system perspective with essential detail. For example:

1. Node architecture overview (see chapter 5) is to unambiguously specify interfaces, connections and constrains;

2. Components Diagrams and Data Model Diagrams is to capture design;

3. State Transition Diagrams is to model dynamic behavior.

3.2 Reverse Architecting Approaches

We have studied some reverse architecting approaches. Some of them cannot handle a complex system nor easily show a complex graph, e.g. Niere et al graphic based approach see section 3.3 Other approaches, and some of them cannot provide an acceptable precision, e.g. Antoniol, Fiutem, and Cristoforetti [22] design pattern based approach, see section 3.3 Other approaches. Workbench Approach does not have those drawbacks, so we study it in this paper.

3.2.1 Workbench Approach

A workbench approach is an open and lightweight environment. It provides an infrastructure for the integration of a wide variety of tools and methods. The open means the workbench allow easy integration of additional tools. The lightweight ensures only necessary dependencies exist among already-integrated tools and data. Based upon the above facts there are no particular static set of tools (extractors, visualization tools, etc.). So the workbench can support many implementation languages; many target execution platforms, and many techniques for architecture analysis.

The workbench approach contains three aspects: View Extraction, Database Construction and Reconstruction. Each aspect can contain any tools and techniques if suitable. For the definitions of those three aspects, see the section 3.2.3, 3.2.4 and 3.2.6.

3.2.2 Dali Approach

The Dali approach is the realization of the workbench environment. The Dali adds an additional activity -View Fusion - to the workbench, so the Dali has four activities: View Extraction, Database Construction, View Fusion and Reconstruction. The current Dali implementation uses an SQL repository for data storage and manipulation. Dali explains how to actually carry on those activities and use the output from each activity from the academic point-of-view. Rational XDE integrates those activities into an application and the user can obtain the architecture information without knowing what is the workbench or Dali approach at all. So the workbench approach is only an environment concept; Dali is a way to realize the workbench approach; Rational XDE is a commercial example in practice. This chapter will use the Dali approach [30] to illustrate the workbench approach. The concrete example will be given in the section 5. The Dali combines a number of commercial and free tools and stores the recovered information in a relational database: the information can be filled and queried by the individual components according to their capabilities.

The architecture to be recovered is the concrete architecture, not the conceptual architecture. Later, it will tell how to check the conformance of the former to the latter, as well as comparing the available documentation, if any, to verify its accuracy. Reference architectures, if they exist, can be used as working hypothesis during Reconstruction. To recover the architecture of an existing system, work is often organized in the following process [15], Figure 2. The arrows show how the information flows among View Extraction, Database Construction, View Fusion and Reconstruction.

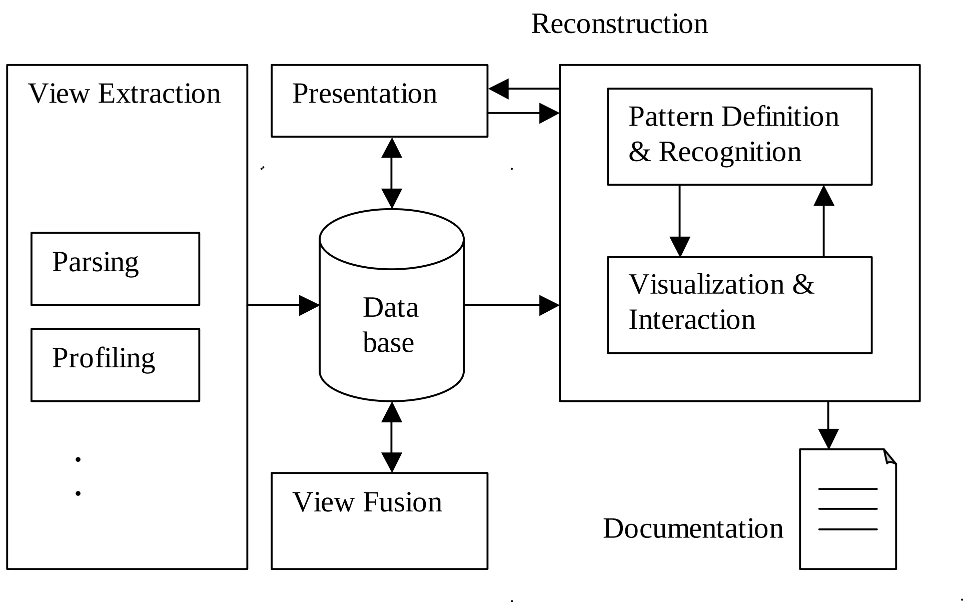

Figure 2. Reverse architecting process

The process is interpretive, interactive, iterative and half-automatic. It needs an actor, e.g. a reconstructor who does reverse architecting, and one or more individuals, e.g. the architect and the software engineer who know the system being reconstructed.

The actor extracts the information from the artifact, then collects and stores them for abstraction of the architecture. During the abstraction, the actor makes a set of hypotheses in order to present and document a proper architecture view. The hypothesis reflects the reverse mappings from the source artifacts to the designed architecture, and ideally it is the opposite of the design mappings. The hypothesis is tested as well; the inverse mappings are firstly applied to the extracted information and then the result is validated. To make it efficient, the actor has to check with the people who are familiar with the system.

3.2.3 View Extraction

View extraction is a workbench activity and a Dali activity. This activity analyzes an existing system design and implementation to be able to build a system model. The output is the information placed in a database, which is used to construct a view of the system in View Fusion.

The extracted information is a mix of the ideal and the practice. The ideal means the information that is mostly wanted to discover the architecture and to meet the goal of reverse architecting. The practice indicates the information that is actually extracted and presented via available tools.

The aim of the activity is to extract information from various artifacts, including the source artifact (e.g. source code, header files, build files) and other artifact (e.g. execution traces). It identifies and captures the interesting elements (e.g. files, functions, variables) and their relationships to obtain several base system views. To retrieve that information, many tools can be used, including profiles and ad hoc (e.g. grep, perl).

The tool imports the artifact, including source files, header files and etc., and then exports the result in a table, like Table 2. Because not all artifacts are importable for tools, the actor can still use the artifact, e.g. architecture documents, for extraction.

ID	Source Element	Relation	Target Element	Description
1	Function	“calls”	Function	Static function call.
2	Function	“access_read”	Variable	Read access on a variable.
3	Function	“access_write”	Variable	Write access on a variable.
4	Directory	“contains”	Directory	Directory contains another subdirectory.
5	Directory	“contains”	File	Directory contains a file.
6	File	“includes”	File	C++ pre-possessor #include of one file by another.
7	File	“contains”	Function	Definition of a function in a file.

Table 2. Typical extracted elements and relations

Like the name of each column shows, the identification (ID) shows the number of each relation, which will be referred later; source and target element indicates the subject and objective; relation is the verb; description is a supplementary. The table should be read like “ ‘source element’ ‘relation‘ ‘target element’.” For example the 5th item, “Directory contains File” means the directory contains a file.

3.2.4 Database Construction

This step is an explicit workbench activity and a Dali activity. Depending on the format of the extracted information, tools (e.g. sed) are available to convert the extracted information to an easy format, e.g. Rigi Standard Format, see Appendix-A. Second, a script (e.g. a perl script [29]) inserts the formatted data in to a database table. Since the relation in Table 2 indicates which relational table to use, the script uses it to find the table and inserts the value. Each relation in Table 2 indicates a relational table (see Table 3 - Table 7), which stores data.

Name	Type	Description
caller	string	The name of the caller.
callee	string	The name of the callee.
visibility	bool	Decide if the item is shown in the architecture view.

Table 3. calls-relation

Name	Type	Description
owner	string	The name of the owner of the variable.
variable	string	The name of the variable.
visibility	bool	Decide if the item is shown in the architecture view.

Table 4. access_read-relation

Name	Type	Description
owner	string	The name of the owner of the variable.
variable	string	The name of the variable.
visibility	bool	Decide if the item is shown in the architecture view.

Table 5. access_write-relation

Name	Type	Description
parent	string	The name of the parent.
child	string	The name of the child.
visibility	bool	Decide if the item is shown in the architecture view.

Table 6. contains -relation

Name	Type	Description
superfile	string	The name of the super-file.
subfile	string	The name of the sub-file.
visibility	bool	Decide if the item is shown in the architecture view.

Table 7. include -relation

Name	Type	Description
elementname	string	The name of each element.

Table 8. elements table

During the construction of the database, an elements table (Table 8), a relationship table (Table 2) and the relational tables are created. Database is able to present the relational tables in views. The process with Rigi format is shown in Figure 3, and it outputs Structured Query Language (SQL) code.

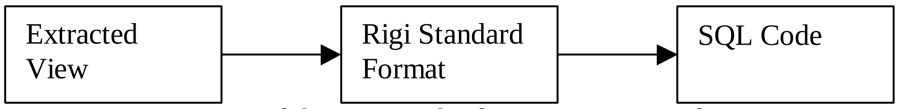

Figure 3. Conversion of the extracted information to SQL format

The selection of the proper database model is important [2].

1. It should be a well-known model, which makes it simple to replace one database with another one.

2. It should allow efficient queries, which is important when source model is quite large.

3. It should support remote access of the database from one or more geographically distributed user interfaces.

4. It supports view fusion by combining information from various tables.

5. It supports query languages that can express architectural patterns.

6. Checkpoints should be supported by implementation, so intermediate results can be saved. This is important in an interactive process because it gives the user the freedom to navigate with the comfort that changes can always be undone.

3.2.5 View Fusion

View fusion is not a workbench activity, but a Dali activity. Because this step can sometimes be unnecessary if the view shown in the previous step is good enough, the complementary information is needed otherwise. The database [2] provides the functionality for reconciliation, augment, and establishment between the elements. However, this step is a Dali specific activity that is not required in the workbench approach. To be able to refine the extracted information, other methods may be used by other specific workbench approach.

There are two kinds of extractions: static extraction and dynamic extraction. The static information is obtained by observing artifacts, while dynamic extraction is received by observing the running system. A function for instance is inherited by a subclass, and is observed during the static extraction. But it can be missed in the dynamic extraction since it is not included in the header file. In case it happens, a SQL query can fuse the two extractions and reconcile the additional information to build a complete and accurate view. The goal of the step is to complete the table and remove as many unnecessary elements in the database as it can.

3.2.6 Reconstruction

This step is a general workbench activity and a Dali activity. At this stage, the information is in a good quality and it is stored in the database. The reconstruction includes two primary activities: 1) visualization and interaction and 2) pattern definition and recognition.

Visualization and interaction enable users to interactively visualize explore and manipulate views. Views are presented in different ways, e.g. a hierarchically decomposed graph of elements and relations.

Pattern Definition & Recognition in Figure 2 provides the definition of architectural patterns and the recognition of such patterns, and documentation is the result from the reverse architecting.

Pattern definition and recognition provides tools for the architecture reconstruction: the definition and recognition of the code manifestation of architectural patterns. Based on the architectural pattern, the actor can export different architectures via various queries. After analyzing each view, the actor can, based on the concept architecture, refine the query to produce a good hypothesized architectural view. However, there are no universal completion criteria for the step and the only useful rule is whether the output is sufficient to support the analysis and documentation.

The workbench uses tools to convert data into an architecture view. The Dali uses SQL to show the architecture view. This view can be large and messy at the beginning because there are many intermediate relations among classes. According to relation partition algebra [5, 6, 7], the actor operates on the architecture view to make the relationships simple. Some relation partition algebra is shown in Table 9. The connector is the valid UML relationship [26].

Rule ID	**Component**	**Connector ->**	**Component**	**Connector ->**	**Component**
1	Class	Generalization->	Class	Generalization->	Class
			Generalization->		
2	Class	Generalization->	Class	Dependency->	Class
			Dependency->		
3	Class	Generalization->	Class	Association->	Class
			Association->		
4	Class	Generalization->	Class	Aggregation->	Class
			Aggregation->		
5	Class	Generalization->	Class	Composition->	Class
			Composition->		
6	Class	Dependency ->	Class	Dependency ->	Class
			Dependency ->		
	…				

Table 9. Some relation partition algebra rules

There are 3 classes, for example Class A, Class B and Class C. The relationship between Class A and Class B is generalization, and dependency between Class B and Class C. Refer to the rule No.2 in Table 9, the Class A will be dependent on Class C and Class B can be hidden in the architecture view.

The SQL query [23] can set the items visibility to “false”, then that item will not be shown in the architecture view, same on the opposite. The Dali approach will use the received data to show the architecture view. Reconstruction should be repeated till the result is acceptable.

3.3 Other approaches

This section compares the workbench approach with another two approaches, Niere et al graphic based approach [11] and Design pattern approach [9, 22], in order to show the advantages of the workbench approach.

Niere et al graphic based approach uses a specialized form of Abstract Syntax Graph (ASG) to encode patterns to a recognized format for the system and, consequently, it is able to show complex patterns (and graphs) in terms of simpler ones just by referencing them, see Appendix-A. An advantage of this technique is that ASG is able to “see through” syntactic variants of the same piece of code. After parsing the code and creating an ASG representation of the system, the analysis proceeds by trying to match simpler graphs first, so that larger graphs (and the more complex patterns represented) that depend on them can later be identified easily. If the analysis had worked in the opposite direction, detection may be considerably more difficult, because the level of detail required specifying complex patterns can be excessive. Since intermediate results are not affected by later calculations, execution can be stopped if desired to browse the work performed so far. But the workbench approach can directly present a complex graph, and then process the simple ones. In another word, the workbench approach can show the complex graph without referencing any simple graphs.

Antoniol, Fiutem, and Cristoforetti [22] introduce a design pattern based approach. The approach firstly maps the source code into an intermediate format, to achieve the independence from the specific tool adopted and from the language of the target, then uses software metrics and structural properties (i.e., the exact types of the classes involved) to reduce the search space, transforming the problem into a search for the shortest path between the nodes representing two classes in a pattern. On average, the precision of this technique is roughly 50%, which is not acceptable. Although we don’t know the exact accuracy of the workbench approach, its accuracy is much higher than 50%.

4 ANALYSIS

4.1 Analysis of the workbench approach

The workbench approach provides a lightweight integration framework whereby tools added to the tool set do not affect the existing tools.

The common problem is that the existing reverse architecting approaches don't clearly show how they are applied in a system. Therefore, Mayrhauser [8] lists the major issues that are interesting to be investigated:

I-1. does the technique extract a useful high-level structure from the system?

I-2. does the extracted architecture facilitate understanding the existing system?

I-3. can the extracted architecture be used in conjunction with existing documentation to leverage understanding?

I-4. what other information is helpful?

I-5. what types of customization of the general approach are helpful?

The workbench approach satisfies all of above issues [I-1 - I-5].

The tool in the approach is language-specific and it is for a large system[1] as well [I-1, I-5]. Currently there is no single tool or tool set that is always adequate to carry reverse architecting out. The tool tends to be language-specific while a number of languages at one time may exist in the artifact. For example Rational XDE has to support C++ language to carry out the case study for the paper. Furthermore, data extraction tools are imperfect. Incomplete or false positive data is often returned [3], so a selection of tools has to be used to filter out the useful information.

The goal of the reverse architecting is a key factor that directs the outcome of reverse architecting. For example, if the node level (see section 5.1.1 for more node definition.) is the goal, the information collected should be on a high level [I-1] and skip the lower level information. The reversed architecture presentation can be used as a basis for documenting the architecture if no documentation exists or it is out of date. It can also recover the as-built architecture, to check conformance against an "as-designed" architecture. This is used to check whether the developer has followed the designed architecture and not eroded the architecture or not broken down abstractions, bridged layers, compromised information and so forth. The outcome of reverse architecting can also be used as a basis for analyzing the architecture or as a starting point for re-engineering the system to a new desired architecture. At last the representation can be used to identify the re-used elements that build an architecture-based software product line.

This approach needs persons to accomplish it [I-2, I-3, I-4]. The current programming language cannot tell the architectural information, which is implemented in different means, i.e. a collection of functions, classes, files, objects and so forth. Neither architectural elements (i.e. "layer" or "connector") nor architectural patterns, if they are used, can be implicitly seen in the source code. In

[1] A large system in this paper means the system consisting of over 10 KLOC.

order to easily discover the architectural information for instance how a system is constructed, the skill and attention from the reverse engineering expert and the architect have to be considered.

4.2 Lessons learned from the workbench approach

Here is a short version of the book [3]. This section gives some lessons to share.

When view extraction is executed; the person should 1) use the least effort extraction, 2) validate the information that has been extracted and 3) extract dynamic information where required.

The least effort extraction consists of the following issues. What information is needed to extract from a source corpus? Is this information lexical in natural? Does it require the comprehension of complex syntactic structures? Does it require some semantic analysis? In each case, a different tool could be applied successfully. In general, lexical approaches are the cheapest to use, and they should be considered if the reconstruction goal is simple.

What and how do persons validate the extracted information? Before starting to fuse or manipulate the various views captured, the actor checks if the correct view information has been obtained. It includes performing detailed manual examination and verification on a subset of the elements and relations against the underlying source code. The precise amount of information to be verified is up to the actor.

Do not miss the dynamic information which exists in runtime linking applications or when the architecture is dynamically configurable.

During the database construction, the database should be simple and the useful intermediate table should be saved in the database. It will be easier to process the view in View Fusion if not build the intermediate table from the extracted relation tables. For instance, if a query is often called, a table can store its result. If the result is queried, that table can be easily accessed. Use simple lexical tools like awk [28] and perl [29] to change the extracted data to a known format to the approach. The database should be carefully designed. For example, what will the primary key be? Are there any database joins to be expensive to implement?

Since fusing views is a Dali specific activity, careful investigation is encouraged. Try different extraction techniques or different instances of the same technique; fuse tables when no single extracted table provides enough information; disambiguate function calls for example if name clashes which is important to be eliminated within the extracted views or to remove the potential confusion by fusing information.

At the Reconstruction, the actor has to iterate the architectural abstraction several times. This is particularly important where there is no explicit documentation. The architect and other stakeholders inspect architectural abstraction, and the comments are used as input of the next iteration. Furthermore, do list all source elements and code segments in a simple way. Especially for the large system, code segments are based on naming conventions and directory structure. The directory structure is used for element aggregations.

4.3 Conclusion

An interesting approach to recovering architectural information is the workbench solution [21]. The conclusion is that the approach needs to integrate different tools to be able to work effectively, as no single tool does the job. Here are results from the following considerations which is a short version of the book [3]:

statically determined information is not enough in many cases, as nearly all modern languages and systems can defer many aspects of a system structure until runtime. On one hand, this is a positive aspect since it provides greater flexibility, but on another hand this creates additional difficulties in recovering the architectural information;

isolated tools suffer from lots of problems. Tools depending on running systems might face with the code that cannot even be successfully compiled due to missing runtime libraries and so on, while parser based tools might have to deal with non-standard language variants and runtime analyzers which are only available on a special system with a certain type hardware;

no architectural information is available at all and, especially in the case of very old systems, and there are no experts to query the missing detail. Therefore, it is important to identify the key architectural constraint and pattern in order to capture the gist of the whole system. Sometimes, it is possible to rely on informal clues provided by the identifiers in the source code, whose importance should not be underestimated [10]. This kind of activity is easier if the work is organized in a comprehensive framework.

Apart from the specific way in which they are combined, there are two alternative types of software analysis, see section3.2.5:

static can detect source code structure and layout and create a map of the dependencies between different parts. Also, in the case of language like C++, it is able to recover information like constant fields names, only available in the source code;

dynamic can determine which approach gets invoked in runtime, because most modern languages, at least partially, defer binding till runtime. It is possible to examine the live behavior of the different data structures, and check what they're used or infer their role in the overall logic of the system. However, there are still some comings to this way since not all valid combinations may be executed using the test cases defined.

Overall the workbench approach is practical. It sounds reasonable on the studied product in chapter 5 and fulfils the issues [I-1 - I-5] presented by Mayrhauser.

5 CASE STUDY ON ONE ERICSSON PRODUCT

5.1 Background

The more complex and variable the system is, the larger the gap between the system and documents normally is. Each non-trivial system should have its own complete documentation. For a product, different customers or customer groups sometimes order the product with slightly different variants, so the varieties of the product increase. It is therefore laborious to record all solutions and hard to completely and correctly understand the complex product. For instance, if a solution is adjusted with subtleties after the architecture design, documents are sometimes not updated. Unfortunately, the only consistency check is the review held before design starts.

The organization has to follow a trend that the developing time becomes shorter and shorter. The 1st release of the product was released some years ago. Since then a variety of new functions, error corrections, change requirements, etc. have been added, and the release 8 is ongoing now. Over years, lots of people have touched the product without recording all changes. Beside the out-of-date documents, the only two ways to learn are 1) to ask the experienced people 2) to read source code. So, it is difficult to fully understand the product since the document does not provide enough information.

Another outstanding phenomenon is that the architecture documentation of the product is not complete. For example, there are both high- and low- level architecture descriptions, but it is still not easy to know the connection between them without referring to the source code. Therefore a complete midway document is needed. This stops engineers from working efficiently and forces them to figure out the information they need.

The above situation actually exists in most software products and therefore reverse architecting is introduced. Reverse architecting is an extension of reverse engineering and it aims to recover the architecture of the product. But the matured technique and tool of reverse architecting are not available yet.

5.1.1 Architecture Information from the Interview

Before the practice in the case study, interviews were individually carried out with Ericsson designers, see Appendix-B.

From the interviews, it is known that the product doesn’t really follow the 4+1 view introduced in chapter 2. The architecture in practice has been improved but the architecture description is out of date. The architecture in this case study has three levels: system level, node level and component level.

The system and node architectures are helpful to illustrate components for newcomers. This case study will focus on the node level that is a bridge between the system and the components.

The highest level is the system level, in which the relation of the product to other external products is described in Figure 4. The document on the system level exists and is almost stable for years. For each latter release, the new functions are just added-on and integrated to the system.

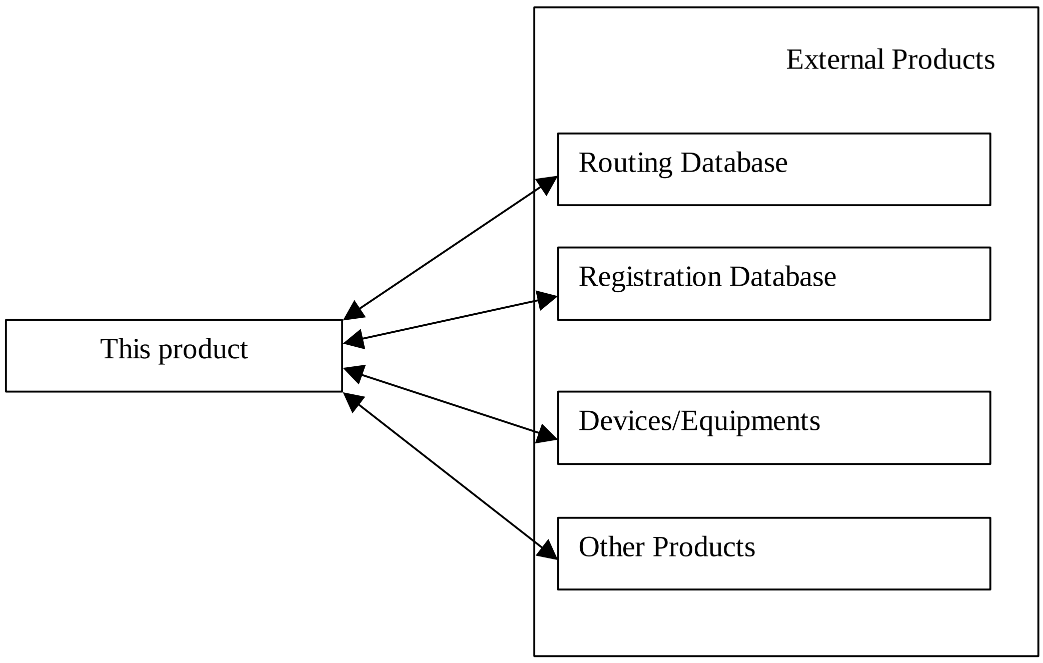

Figure 4. System architecture diagram

Figure 5 shows the node architecture diagram. The product in the case study consists of 5 core components (A, B, C, D and E). Component A, B, C and D are dependent on Component E that acts as a communication carrier for them. The process in this level is that a node architect transfers requirements into a node architecture, which contains software elements, properties and their relationships, and documents it in a node architecture description. A node architecture description can also have the key concept, the non-functional functionality, and the company and design policy. Unfortunately, the information in this levels description is not sufficient.

In the component level each component consists of a set of classes and the interfaces. Normally the component is well documented. But if many changes are still accepted after the document is inspected, it will probably not include those new changes. Since this level is not too deep for a node architect, we don't show its diagram here.

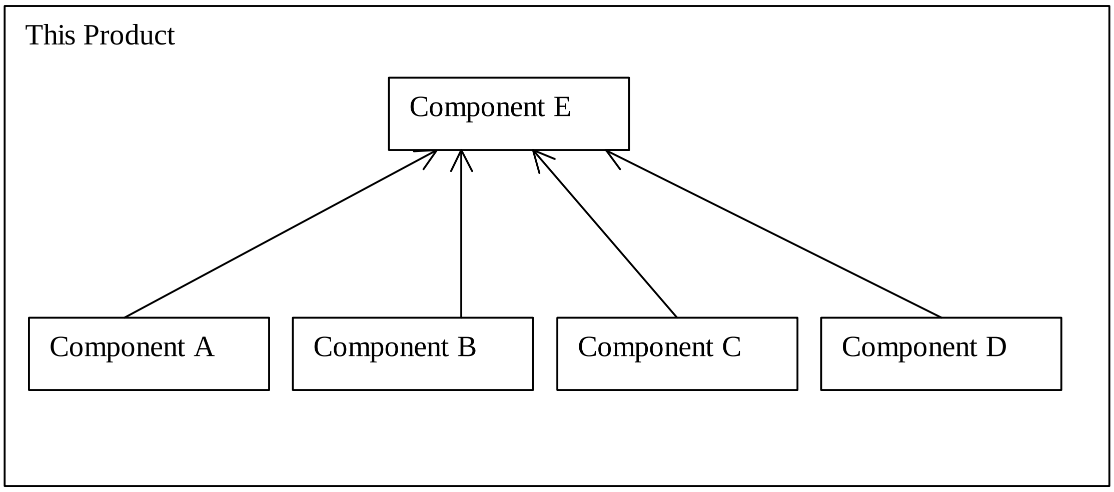

Figure 5. Node architecture diagram

5.2 Approach

One reason to select Rational XDE [25] is that it is a licensed workbench implementation, because some free tools mentioned in the section 3.2 are not so stable as that of Rational XDE and are in some cases not free at all for non-commercial use. Rational XDE is partial open. Due to the fact that it is a commercial application, it cannot be complete open. But it provides a feature – Extended XDE that enables the user to integrate any tools to XDE or modify the data in XDE. Rational XDE is not lightweight that is against the definition of the workbench. Since the most commercial tools are heavyweight and require the source should be in a certain formats, we assume Rational XDE is lightweight from the commercial point-of-view.

Secondly, Rational XDE is a commercial application that automatically carries out the workbench approach activities, i.e. View Extraction, Database Construction and Reconstruction. If Dali approach is selected, the user has to study lots of tools and manually analyze the input and output of each activity. The time to use Dali to run reverse architecting is long; and the executants and the challenge of the project can extremely affect the quality of the result of reverse architecting. While Rational XDE don't have these kinds of personality affects.

Here is an example on how Rational XDE reverses C++ source code. Rational XDE reads in the C++ source code for the product, and then, according to the file structure how the source files are stored, automatically saves the data and the analysis of the data to its internal database. The analysis is each components class diagram (e.g. Figure 6) and some intermediate data mentioned in the Workbench approach. Rational XDE cannot automatically reverse the input to complete node architecture, because some factors (e.g. the node architecture diagram, non-functional functionality, use-case, etc.) are missing. The internal database provides the standard interfaces that the SQL can use. If use Extended XDE to make another application, the user can use the data in the database and the external information automatically get the missing part for the node architecture, or refine the data in the database and so on. Of course the user can always manually make the missing node architecture.

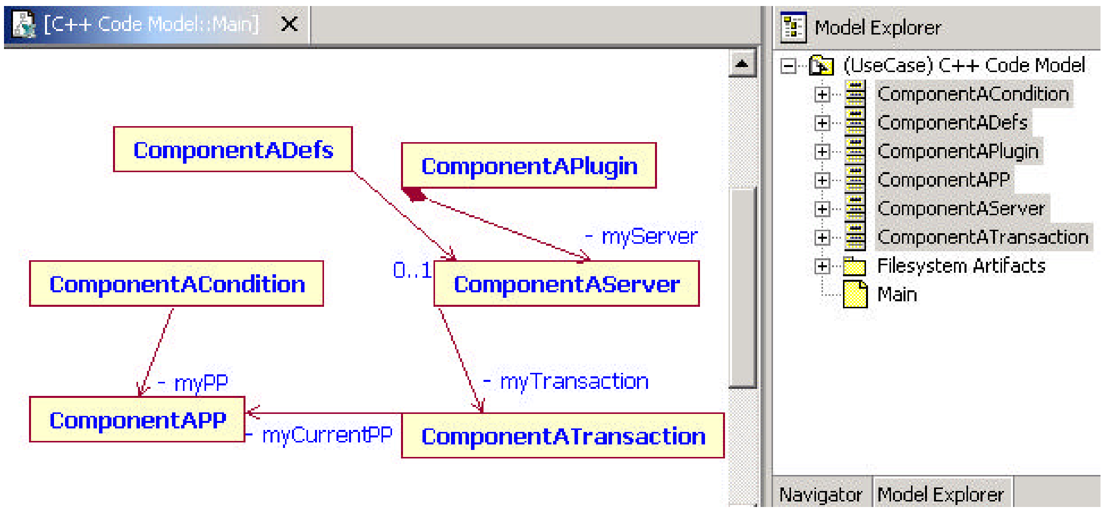

Figure 6. Class diagram for one component

5.3 Lessons Learned from the Case Study

The table below lists the result of this case study. The column item lists what we compare between the existing node architecture description and the output from XDE. “Yes “ or “No” means if it is true for the current item. The designer can use most of the Rational XDE outputs do reverse architecting.

Item	**Existing node architecture description**	**Output from XDE**
Generate class diagrams	Yes	Yes
Follow the UML Standard	No	Yes
Affect the development strategy (e.g. to do proof of Rational XDE technology)	No	Yes
Automatically generate the node architecture diagram	No	No
Generate the physical and deployment views and use cases	No	No
Generate outstand Key concepts (e.g. the key components) and their relationships	Yes	No
Output Company/Design Policies (e.g. how to design the product)	Yes	No

Table 10. Result of the case study

The class diagram from XDE provides more information than those in the existing node architecture description. It takes XDE around 2 ~ 5 minutes to generate them from the source code, so it is easily to use XDE to make the latest class diagram. Furthermore the diagram strictly follows the UML 2.0 standard that is universally understood and the class diagram is exactly based on the current implementation.

On the other hand XDE cannot automatically generate what we are interested in most - the node architecture diagram. If with Extended XDE there is a chance to automatically make it. But Extended XDE is a component specific tool and it has to be synchronized with any changes from any components. So no matter with or without XDE the designer has to do more than what they expect. Therefore XDE cannot ease the maintenance of existing system or speed up the architecture design in the next release.

Rational XDE affects the development strategy. The organization has to allocate the money for the evaluation of the Rational XDE technology and the education on Rational XDE. If decides to use Rational XDE, the organization has to plan how to deploy it in a project and may change the organization structure. As a Windows application, Rational XDE is useless in the UNIX environment.

XDE can neither generate the key concept and their relationships that are useful for the abstraction reusability, nor the physical and deployment views and use cases. The designer has to do the extra work to make them since they actually are very important for designers to know and make early design decisions.

XDE doesn’t show the company/design policies. The policy can be the component level issue (e.g. Domain issues) and the node level issue (e.g. Resource Access), and the mandatory standard issue (e.g. protocols). The policy is not reflex from the source code that XDE requires, so XDE won’t be able to find the information. The policy is important for designers to design and implement the architecture in a right way. If nowhere shows those policies, it is a risk that the product will be developed in an unexpected way.

So Rational XDE is not an application that automatically reverse the input to the node architecture, but a good application for the components level because it can easily output the latest components information. If XDE is the only choice to do the reverse architecture, Extended XDE can be an assistant to help the user realize it.

With the development of the technology, it can be true that all architecture information one day will be recovered from the artifact by a reverse architecting approach.

6 CONCLUSIONS

The architecture of a product can be compared to its backbone. It is the key element in ensuring the product to adapt changes but still be comprehensible and manageable. Thus, it is of importance to recover the architecture whenever the documentation becomes obsolete, as the alternative is scrapping the whole system.

As the reverse architecting field is comparatively young, there is currently no simple technique that works in all circumstances (in fact, such a technique might not even exist at all on a theoretical level), let alone one considerably better than the rest. Some approaches depend on the reference architecture to compare against; some depend on the possibility of building and running the system; some depend on the language used to implement the system; some are so inaccurate that manual reconstruction is preferred. Since the reverse architecting field is not mature, it can work better if engages human knowledge. Though strictly speaking architectural assessments are not a reverse architecting activity, they play an important role in the grand scheme of software evolution. Tools assess the activity can only be improved, as they are currently missing.

The workbench approach is an attractive solution. It is efficient to recover the lost architecture and improve the quality of the product, see chapter 4. Although at the beginning the workbench approach can be hard to follow, it eases the future releases of the product. Especially, it will benefit the un-documented product.

The workbench approach normally is complicated and advanced. This approach includes different kinds of activities to generate the node architecture view. The additional knowledge, e.g. database knowledge, script knowledge and etc., is required. Rational XDE do the workbench approach activities but not generate all necessary artifacts to do reverse architecting.

7 REFERENCE

1 Jan Bosch, Design and Use of Software Architectures, Addison Wesley, ISBN 0-201-67494-7, 2000

2 Len Bass, Paul Clements and Rick Kazman, Software Architecture in Practice, Addison Wesley, ISBN 0-321-15495-9, 2003

3 Paul Clements and Linda Northrop, Software Product Lines, Addison Wesley, ISBN 0-201-702332-7, 2001

4 Anders Henriksson, Henrik Larsson, **A Definition of Round-trip Engineering**, Department of Computer and Information Science, Linköping Universitet

5 Alexandru Telea and Alessandro Maccari and Claudio Riva, **An open visualization toolkit for reverse architecting**, Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02), 2002

6 André Postma, Marc Stroucken, **Applying Relation Partition Algebra for Reverse Architecting**, Workshop Software-Reengineering, Bad Honnef, 27-28. May 1999

7 L. Feijs, R. Krikhaar, R. van Ommering; "**A Relational Approach to Software Architecture Analysis**", Software Practice and Experience, vol. 28, 4(April 1998)

8 **A. von** Mayrhauser, J. Wang, M.C.Ohlsson and C. Wohlin, **Deriving a Fault Architecture from Defect History**, 10th International Symposium on Software Reliability Engineering, November 01-04, 1999**,** Boca Raton, Florida

9 G. Antoniol, R. Fiutem, and L. Cristoforetti, **Design Pattern Recovery in Object-Oriented Software**, Proceedings, 6th International Workshop on Program Comprehension, Jun 1998.

10 Ted J. Biggersta, **Design Recovery for Maintenance and Reuse**, Computer, 22(7), Jul 1989.

11 J. Niere, **Fuzzy Logic based Interactive Recovery of Software Design**, Proceedings of the 24th International Conference on Software Engineering (ICSE), USA, ACM Press, May 2002.

12 Premkumar T. Devanbu, David S. Rosenblum, and Alexander L. Wolf. **Generating Testing and Analysis Tools with Aria**, ACM Transactions on Software Engineering and Methodology (TOSEM), 1996.

13 Thomas A. Corbi, **Program understanding: Challenge for the 1990s**, IBM Systems Journal, 1989.

14 John Vlissides, **Reverse Architecture**, Dagstuhl Software Architectures Seminar, Aug 1995.

15 René L. Krikhaar, **Reverse architecting approach for complex systems**, 1997 International Conference on Software Maintenance (ICSM '97) October 01 - 03, 1997 Bari, ITALY

16 Galal H. Galal-Edeen, **Reverse Architecting: seeking the architectonic**, Ninth Working Conference on Reverse Engineering (WCRE'02), October 29 - November 01, 2002**,** Richmond, Virginia

17 Dilip Soni, Robert L. Nord, and Christine Hofmeister, **Software Architecture in Industrial Applications**. International Conference on Software Engineering, 1995.

18 Elliot J. Chikofsky and James H. Cross II, **Reverse Engineering and Design Recovery: A Taxonomy**. IEEE Software, _17, Jan 1990.

19 Shane Sendall and Jochen Kuster, **Taming Model Round-Trip Engineering**, Computer Science Department, IBM Zurich Research Laboratory, 2004

20 Philippe Kruchten, **The 4+1 View Model of Architecture**, IEEE Software, Nov 1995.

21 Steven G. Woods, S. Jeromy Carrière, and Rick Kazman, **The Perils and Joys of Reconstructing Architectures**, SEI Interactive, Sep 1999.

22 G. Antoniol, R. Fiutem, and L. Cristoforetti. **Using Metrics to Identify Design Patterns in Object-Oriented Software**, Proceedings. 5th International Software Metrics Symposium, Nov 1998.

Online Documents

23 SQL web site: http://www.w3schools.com/sql/sql_intro.asp, 2004

24 Ericsson web site: http://www.ericsson.com/, 2004

25 IBM Rational web site: http://www-306.ibm.com/software/rational/, 2004

26 UML website: http://www.uml.com/, 2004

27 Rigi group, http://www.rigi.csc.uvic.ca/rigi/manual/node52.html, 2004.

28 Sun website: http://www.sun.com/, 2004

29 Perl website: http://www.perl.com/, 2004

30 The Dali Architecture Reconstruction Workbench, http://www.sei.cmu.edu/ata/productsservices/dali.html, 2004

APPENDIX-A TOOLS

Rigi Standard Format

Rigi Standard Format [27] is Rigi's own file format and is also used by so many third party tools that it can be considered a de-facto market standard of sorts.

There are two major variants of RSF, unstructured and structured: the most obvious and visible difference between them is that lines in unstructured RSF files are triples while lines in structured RSF files are quadruples, with the extra element being used to store information about subsystem hierarchy and layout. Since documentation about the structured variant is scarce and, anyway, most tools only use the unstructured variant, we'll describe this one alone: each triple consists of a verb, that must be listed in a Rigi domain to be allowed, and two other elements which can either be a source and a destination node (in the case of arcs) or a node and its attribute.

Abstract Semantic Graph

An Abstract Semantic Graph [12] aims at being complete and language independent. ASG is essentially an abstract semantic text with embedded semantic information. Specially a reference to an entity is represented by and edge pointing to a simple leaf node that holds the name of the entity. In an ASG, a reference is represented by an edge pointing to the root of the (shared) graph in the ASG that represents the declaration of the entity.

The ASG graphs differs from a general graph in that both edges and nodes are typed, according to the model specifications, and they represent entities and their properties.

APPENDIX-B INTERVIEW RECORDS

I interview the Ericsson designers in May, 2004.

They introduced what the current status of their product and what they expect. Each interview lasts around 30 – 60 minutes and is recorded in an MD disc.

The interviewed questions are:

Q-1. Which kind of backgrounds do you have in the software product?
Q-2. What are your opinions on the architecture of the product?
Q-3. Is your description complete from the different levels of the whole system point-of-view?
Q-4. Based on what do you define the architecture you just described? Is it the correct principles? Why not take some other simple or advanced architecture principles?
Q-5. Are there any existing architecture descriptions of the product?
Q-6. What do you expect from the architecture descriptions?
Q-7. Are there any activities to verify the architecture? E.g. How to know the difference between the architecture designed and the one implemented?
Q-8. What are important and should be improved?

1. Interview 20040518 – #1

X starts in the year 2001 and has been involved in 3 releases. This product has a good architecture. In the system level this product uses business products and standard protocols, e.g. Oracle DB, XML. This ensures the quality on the higher level. Some information in the next level (node level) is missing. For example it would be good to have a complete node architecture description that describes the components and their connections. It nowadays in text elaborates the key concepts and their relationships and company and design policy. When he started, the only way to learn the product was to read the source code. The lowest level is component level that has very good artifact.

The verification between the designed architecture and the implemented architecture is done in an efficient but informal way. When they have coffee, they talk about the implementation and design, and fix the flaws if there are.

He suggests that the product should firstly study the system level then the node level. The node architecture should be as simple as possible, but not simpler. The architecture description should be easy to understand.

2. Interview 20040518 – #2

X started some years ago and has been participate many releases. The architecture shows a main task about how is solved by functions. The functions are clearly and possible to be seen and reused by other products. This product has a good architecture in practice. This structure is complete, correct and simple which is the most important. Basic node test, part of in the Ericsson design process, mainly verifies the designed architecture and the implemented architecture.

Unfortunately, a complete node architecture description doesn't exist. Since the functionality is some kinds of add-on to the previous releases, this product follows the old architecture on a high level. The architecture description should explain the relations and cope with the high level architecture. It should have the traffics, flows, operation and maintenance functions, etc.

3. Interview 20040526 – Node Architect and Designer #3

X started as a designer of this product in the year 2002. The highest level is the system level, in which relations to other external products in the telephone network are described. The next level is node level. Although a node architect works on the requirements, an up-to-date node architecture description does not exist. The lowest level is the component level for which the documents are often well maintained. But if many change requests still come after the documents are inspected, they will not be updated in time.

The product consists of 5 core components (A, B, C, D and E). Component A, B, C and D don’t directly talk to each other. Instead they use Component E as a carrier to transfer the information.

The current node architecture in practice is not so bad. But the artifacts are not in the same steps as that of the product in practice.

4. Interview 20040611 – #4

X started from the 1st release of the product. The product hasn’t reached the aimed architecture, and each release includes some improvements towards the goal. There are lists to record action points. The document becomes complete with the mature of the product. It can be assumed that the product will be possible to maintain a good documentation store in the future. This case study can provide them with a basic architecture for the release.