Agile Architecture Recovery

by

Davor Svetinovic

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2002

©Davor Svetinovic 2002

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

// Abstract

Many software development projects start with an existing code base that has to be tightly integrated into a new system. In order to make a robust system that will achieve the desired business goals, developers must be able to understand the architecture of the old code base and its rationale.

This thesis presents a lightweight approach for the recovery of software architecture. The main goal of the approach is to provide an efficient way for architecture recovery that works on small to mid-sized software systems, and gives a useful starting point on large ones. The emphasis of the process is on the use of well established development techniques and tools, in order to minimize adoption costs and maximize the return on investment.

For my mom, dad, and sister

Acknowledgements

I would like to thank my advisor Michael Godfrey for his invaluable advice and support. Mike's instructions and guidance were of the great help for the successful completion of my degree.

I would like to thank my thesis readers, Daniel Berry and Anne Pidduck, for their comments and suggestions. They greatly contributed to the quality of the thesis. I would also like to thank Dan for an excellent collaboration and supervision of my teaching assistant work.

My studies and research were generously supported by Fonds Québécois de la Recherche sur la Nature et les Technologies and University of Waterloo. I am thankful to these two organizations.

Finally, many thanks to my family, girlfriend, and close friends for their support, understanding, and encouragement.

Contents

1 Introduction **1**
1.1 Motivation and Problem Description: Project Development Issues 1
1.1.1 Conventional Process Issues 2
1.1.2 Unified Process 4
1.1.3 Agile Development Processes 5
1.2 The Proposed Solution and Major Thesis Contributions 8
1.3 Thesis Organization 10

2 Key Concepts and Related Work **11**
2.1 Software Architecture and Design 11
2.1.1 Architectural Views 12
2.1.2 Reference Architecture 13
2.1.3 Architectural Refactorings and Repair 14
2.1.4 Architecture Reengineering and Visualization 14
2.2 Quality Attribute Theory 14

3 Agile Architecture Recovery Process **17**
3.1 Architecture Recovery Issues and Goals 17
3.2 Architecture Recovery Process 19
3.2.1 Architecture Meta-model 21
3.2.2 Architecture Recovery Techniques 24
3.3 Architecture Recovery Process Steps and Legacy Systems 32
3.4 Summary 37

4 Case Study: X MultiMedia System **38**
4.1 XMMS: Introduction 39
4.2 XMMS: Architecture Recovery 39
4.2.1 Project Elaboration 39
4.2.2 Architecture Recovery: First Iteration 41
4.2.3 Architecture Recovery: Second Iteration 53
4.3 XMMS: Focused Evolution Recovery 64
4.4 Summary 69

5 Evaluation **70**
5.1 Case Study Validity 70
5.1.1 System Concerns 71
5.1.2 Analyst Concerns 72
5.2 PBS: Reference Process 73
5.3 Project Management Issues 75
5.3.1 Process Iterations 75
5.3.2 Individual and Team Issues 76
5.4 Process Step Issues 77
5.4.1 Project Elaboration 77
5.4.2 Analysis of Existing Artifacts 78
5.4.3 Domain Architecture 78
5.4.4 Conceptual Architecture 79
5.4.5 Concrete Architecture 80
5.4.6 Architecture Rationale 81
5.5 Process Technique Issues 83

6 Conclusions and Future Research **84**

List of Tables

4.1 XMMS: Basic System Metrics . . . 40
4.2 General System Features . . . 42
4.3 User Interface and Visualization Features . . . 42
4.4 Plugins Features . . . 43
4.5 Other Features . . . 43
4.6 Iteration 1: Actor-Goal List . . . 44
4.7 Iteration 1: Goal-Process List . . . 45
4.8 Iteration 1: Process-Resource List . . . 45
4.9 Iteration 1: Business Rules List . . . 45
4.10 Iteration 1: Presentation(UI) Layer Subsystem Responsibilities . . . 48
4.11 Iteration 1: Application Logic Layer Subsystem Responsibilities . . . 49
4.12 Iteration 1: Service Layer Subsystem Responsibilities . . . 49
4.13 Iteration 1: Architecture Rationale . . . 52
4.14 Method-Level Responsibilities (Main Module) . . . 60
4.15 Module-Level Responsibilities . . . 61
4.16 Iteration 2: Architecture Rationale . . . 65

List of Figures

1.1 Waterfall Model 2

3.1 Major Architecture Recovery Activities 20
3.2 Architecture Meta-model 25
3.3 PBS: Architecture of Linux Operating System 30
3.4 Architecture Recovery Artifacts and Relationships 33

4.1 Iteration 1: Static Conceptual Architecture 47
4.2 Iteration 2: Static Conceptual Architecture 54
4.3 PBS XMMS Top Level View 56
4.4 Visualization Subsystem 57
4.5 Input Subsystem 58
4.6 CDAudio Subsystem 59

Chapter 1

Introduction

The importance of reverse engineering and software architecture has been recognized in all areas of software engineering that deal with legacy software. Much effort has been invested to improve our ability to understand, present, repair, and improve these old mission-critical systems.

Recently, another motivation for the use of reverse engineering techniques and software architecture has appeared. This is the emergence of a new set of development methodologies — agile development processes.

1.1 Motivation and Problem Description: Project Development Issues

The first response to the problems introduced by the chaotic "code and fix" style of development, was the introduction of the development methodologies that have been inspired by the methodologies of other engineering disciplines, such as civil and mechanical engineering. Development in these other fields consists of an *engineering* phase followed by a *construction* phase. The engineering phase is characterized by detailed planning of all the aspects of the construction phase, which includes, among other things, a detailed design of the product to be built. The main activities performed during the engineering phase are the analysis of the problem and the design of the solution. The engineering phase usually costs much less than the construction phase. For example, in civil engineering, the engineering phase typically costs 10-20% of the whole development

process [Fow]. Also, most of the creative and intellectually challenging activities are performed during the engineering phase. On the other hand, the high level of automation and high demand on manual skills characterizes the construction phase.

These conventional development methodologies were applied to software development with the hope that one will achieve similar characteristics and stability in software development. A typical model that describes this way of software development is the waterfall development model [Roy98], Figure 1.1.

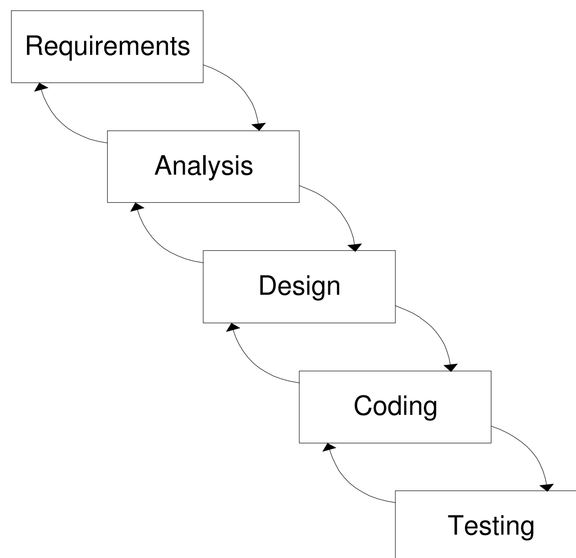

Figure 1.1: Waterfall Model

1.1.1 Conventional Process Issues

As depicted in Figure 1.1, the development activities of the waterfall model are performed sequentially. The goal is to perform, complete, and document each activity before the next one is started. Experience has shown that this traditional development suffers from serious problems, which have caused a high rate of unsuccessful projects. These problems include:

- early freezing of requirements,

- late integration and deployment,
- late risk resolution,
- lack of validation and feedback,
- lack of support for evolving requirements and systems, and
- documentation overhead.

Not only it is very hard to discover all the requirements for a complex project at once, but for most software problem domains requirements change very often. This affects the development in different negative ways that result in the dissatisfaction of all parties. The most negative effect is that the product built at the end does not conform to the customer's real needs (*i.e.,* the requirements specification did not match the customer's real needs) or does not perform as it was supposed to (*i.e.,* the requirements were reasonably well captured but were not implemented correctly). The solution to this problem has to be looked for in a way of organizing the development process such that developers get constant feedback from the customers on what they have to build and how, and accommodate changes due to customers' needs. We will discuss later how modern processes address this problem.

Large, complex software systems are built from many separately developed modules, and often have a very complex deployment process, especially in distributed environments. Practice has shown that insisting on the development of all modules and then performing the integration and deployment often leads to serious incompatibility problems, and in many cases to the complete breakage of the system. The solution to this problem is in integrating and deploying parts of the system as soon as possible.

Most of the decisions considering project aspects such as cost estimation and schedule planning are performed in the early phases. These estimates and other risk predictions have shown to be wrong in most cases [Sch98]. In projects that follow the waterfall development model, these false predictions show very late, during the coding and later phases. This leads to financial losses, abandoned projects, and so on. The solution to this problem is in a way of assessing risks based on the actual project progress. Since the progress is shown best during the coding phase, one should look for a way of distributing project activities in a way that some implementation is done as soon as possible.

The various forms of documenting different steps and artifacts during the software development process have become so incorporated and overemphasized in the project that the time and cost overhead in producing them has overcome their value. Also the usefulness of this documentation is questionable since it becomes outdated very quickly and is not used frequently by the development teams. The reason for this is in the nature of the software development. Built software artifacts change so often that it is not economical to keep them documented using traditional documentation processes. The solution to this problem is in the identification of the software documentation that is actually used during the development, and in its automatic production and updating.

1.1.2 Unified Process

As the response to the previously mentioned problems, several modern development processes have appeared during the 1980s and early 1990s in order to address these problems. In the mid-nineties, the Rational Unified Process [RUP] was developed with the goal of unifying the best ideas of this new group of development processes. It is highly customizable to specific production environment needs. We are interested in the general principles and practices on which the framework is based.

The main characteristics and principles of RUP can be summarized as [Krub, Roy98]:

- Iterative and incremental
- Architecture-centric
- Object-oriented and component development based
- Managed and controlled
- Highly automated

Iterative and incremental — An iterative life cycle is based on successive refinement of the system through multiple cycles of analysis, design, coding and testing. In every cycle, we tackle only a small set of requirements, which results in incremental growth of the system. So after each cycle, more and more functionality is added to the system. This approach has several advantages:

- The complexity the system development is not overwhelming since the system is broken down into more manageable units.
- The shorter iterations allow earlier system integration.
- Earlier feedback from the end users is possible since the parts of the system can be used while the rest of the system is still in the development.
- Requirements can be adjusted and changed more easily during the development process.

Architecture-centric — Architecture-centric means that the architecture of the system is established early in development process, and is used as a driving force throughout project. A well designed architecture tends to stay stable throughout the initial development and changes little as the system subsequently evolves over time.

Object-oriented and component development based — The process emphasizes the use of object-oriented and component technologies, as opposed to procedural paradigm and all-in-house development.

Managed and controlled — The process specifies and insists on the strict project management techniques and milestones.

Highly automated — The goal is to have all the documentation automatically produced, all the build artifacts tightly integrated, and tools for full process support in order to automate repetitive, error prone, and tedious tasks.

The design techniques used in the process are semi-formal. The most commonly used language for expressing design artifacts is Unified Modeling Language (UML) [FS99, Lar01].

1.1.3 Agile Development Processes

As we have seen, project development processes have moved from the chaotic “code and fix” style, through highly methodological waterfall model based development processes to the iterative, modern processes that deal with certain drawbacks of the waterfall approach. Recently, several other methodologies have appeared to solve the problems that still exist in the modern processes. These processes are commonly called agile processes. The best known agile development process is Extreme Programming (XP) [Bec99, JAH00].

The first distinction, which is easily seen in these new processes compared to the older ones, is in the amount of management work. Followers of the new lightweight processes have commonly claimed that highly structured processes are too bureaucratic. Management overhead is the source of a lot of problems in all previous processes. This is also the reason why they are called heavyweight and these new ones lightweight.

The reason for this distinction lies in the core values and ideas of lightweight processes [Fow]:

- Lightweight methodologies are *adaptive* rather than *predictive*.
- Lightweight methodologies are *people oriented* rather then *process oriented*.

If we analyze again all previously presented processes, we remark that planning, analysis and design dominate the beginning of every one of them. This was much more emphasized in waterfall-based processes than in iterative ones. Iterative processes have tried to distribute these activities over the whole development lifecycle because of previously mentioned problems. This was not enough to the followers of lightweight processes, who argued that in practice this causes developers to plan activities in advance, to predict behavior and changes of the process and the product ahead of time, when the basis for these predictions is not strong enough. This way of reasoning and planning introduces high amounts of complexity and actually slows down the development. It is difficult to predict what will be needed and what will happen in the future. All this has resulted that these previous processes are in practice resistant to changes. This has had further effects on how the development business was led. For example, development companies have been trying to freeze requirements as soon as possible and have been resistant to change so that the customer has actually suffered if the requirements change while the product is still in development.

To deal with this problem, lightweight processes try to minimize prediction. The idea is to do exactly what the customer needs at this point and to adapt to any new needs as they arise. This adaptation is not done by trying to predict what the customer might need and to build a complex framework in order to accommodate changes, but by restructuring the existing system to accommodate the changes when they are actually identified. Old processes have been based on the assumption that it is easier to build to support the change in the future than to have to change the system to accommodate changes at that point in time. Practice has shown that this is often not the case, and therefore this new opposite philosophy that lightweight processes support.

It is this need for constant restructuring of a system to accommodate changes that causes a need for constant use of architecture recovery techniques and principles during development. Also, a practical non-existence of forward design activities enforces this need for re-engineering even more.

The other core difference between heavyweight processes and the lightweight ones is in the treatment of the developers from the management perspective [Fow]. It was remarked, in practice, that developers are treated as pure resources within the development process. This has its roots in the conventional engineering disciplines where most of the decision-making is performed by a small group of architects and engineers, while the majority of other workers are required to have only manual skills. This skills distribution has worked fine in these disciplines due to the nature of the development. Proponents of the agile development philosophy have remarked that this does not work very well in software development. The inter-dependency of the artifacts produced during development is so high that all developers have to be proficient in all skills. Reeves argues that coding is the part of the system's design which leads to the conclusion that the software construction is almost completely automated, and thus that we only have engineers working in the software development [Ree]. Lightweight processes take the radical approach in putting people in the first place in the development. Among these skills that all developers have to be proficient in is architecture recovery. This also implies that these activities should be performed constantly and by everyone on development team. Our recovery process is based on this community effort, as discussed later.

These two principles lead to the much lower need and amount of bureaucracy in the development. The majority of people are concentrated on actual development instead of managing.

One of the most controversial aspects of Extreme Programming (XP) and other lightweight development methodologies [Fow] is an almost complete rejection of formal and semi-formal upfront design activities. This rejection is backed up by the arguments that these design activities do not work very well in practice due to a variety of technological, management and human factors:

- Most developers like to concentrate exclusively on programming.
- Design documents are almost never in synchronization with code.
- During active development, a lot of time is spent updating design documents.

- The produced documents are not used much by programmers.
- Tool support for roundtrip engineering is not good enough.

Instead of these rejected formal and semi-formal design activities, lightweight processes depend on:

- Use of CRC cards [Coc] for initial, quick design.
- No design documentation. Code and programming conventions are considered to be sufficient for design recovery. CRC cards are not kept.
- Use of a system metaphor for design abstraction purposes.
- Simplicity and adaptive design approach — Simple design and refactoring make systems easier to change to accommodate new requirements than to try to predict them and build an infrastructure to support them [Fow].
- Continual refactoring [Fow99].

In summary, the core principles of new agile methodologies have created a need for efficient architecture recovery as a crucial part of the whole process of development of a new system. This is in addition to the already existing need for the architecture recovery of legacy systems.

1.2 The Proposed Solution and Major Thesis Contributions

In this thesis, we present a systematic approach for recovery and presentation of a software architecture and its rationale (*i.e.,* the underlying reasons for that particular architecture) for small to mid-sized systems.

Our goal is not to invent yet another architecture recovery process, but to build on existing ones and to improve their applicability for the architecture recovery of systems that are built using agile development processes, and to conform to agile development philosophy. Two main properties of such systems are their moderate size and changing requirements in their problem domain. Also, the particular emphasis is on the recovery of the architecture rationale, as design decisions made during development and requirements that led to them are not documented and

preserved. Architecture rationale recovery provides us with answers to why architecture was designed or has drifted in a particular way.

It is not the goal of the thesis to present the definitive set of activities to be performed and artifacts to be built. This would be contrary to the inherent difficulties of software development, properties of different problem domains, different circumstances under which recovery is performed, and purposes for which artifacts are produced. The goal is to lay out a theoretical background for the recovery process and clearly define the constraints to be satisfied. This is done through the use of different meta-models, such as problem domain and architecture meta-models, and the analysis of the conformance of different artifacts to the properties imposed by the agile constraints. Following the theoretical introduction, we propose a set of recommended activities and artifacts to be produced during architecture recovery. We apply the approach for architecture recovery of a mid-sized software application to evaluate its effectiveness, usability, and applicability.

The main contributions of this thesis research are:

- Agile architecture recovery process — The thesis includes a complete description and an application of one possible instance of a process that can be used directly for the recovery of software system architectures. The main goal of the process is on being an agile and efficient solution for the architecture recovery of small and medium-sized software systems. It can also be used for the recovery of the architecture of small and medium-sized subsystems of larger software systems, and the results can be integrated using the techniques from other processes that are used to deal with large software systems.

- Techniques for the recovery of architecture rationale — The particular emphasis of our architecture recovery process is on the recovery of architectural rationale (*i.e.,* decisions and influences that have shaped the architecture of the system). We propose a set of techniques and heuristics, as a part of our process, which allow us to recover and estimate rationale behind the software system's architecture.

- Architecture recovery of a multimedia system — A detailed architecture recovery of a multimedia system is presented. The contribution is twofold:

  1. it serves to validate the proposed architecture recovery process and

2. it enlarges the body of knowledge about multimedia software applications and their architecture.

1.3 Thesis Organization

The rest of this thesis is organized as follows. In chapter 2, we introduce related work in software architecture, software design, and attribute theory. In chapter 3, we present our architecture rationale recovery process and all related techniques. In chapter 4, we present a detailed case study of architecture recovery of a medium-size software system — the XMMS multimedia system, a 65 KLOC C program. Chapter 5 presents an evaluation of the process including the analysis of process drawbacks and possible enhancements. Finally, chapter 6 discusses conclusions and future work.

Chapter 2

Key Concepts and Related Work

In this chapter we present background and related research on software architecture, design, and attribute theory, emphasizing how it relates to our research that is presented in the rest of this thesis.

2.1 Software Architecture and Design

One of the main tools that has been used in all areas of software engineering to solve different problems is abstraction. Software architecture is a recent significant result of its use. It helps engineers to conquer the complexity of the system by giving them the ability to organize, understand, present, and manage system in a much more natural and easier way, than they can just do by using programming language features.

Currently, there is no universally accepted definition of what software architecture is. Nevertheless, common to all proposed definitions is that architecture deals with large scale constructional ideas and techniques — high level organization of system, constraints, motivation and rationale, architectural styles, interconnections, and collaborations [PW92, SD96, BCK98].

Software architecture research can be categorized in following groups, which together form a body of knowledge and techniques that software architects use during development:

- Architectural styles and design — study of proven architectural designs and practices that help architects design flexible and extensible software systems.

- Component-based technologies — different technologies that are used to build systems using proven architectural practices.
- Architectural recovery, visualization, and analysis — the focus of this research is on increasing system's understandability, reliability, and reusability.
- Run-time system generation and manipulation — the study of different techniques for run-time system modularization and modifications.

The focus of our research is on architectural recovery, visualization, and analysis. The following architectural topics and techniques are the main tools used by researchers in this area:

- Architectural views
- Reference architectures
- Architectural refactorings and repair
- Architecture reengineering and visualization

2.1.1 Architectural Views

The biggest problem in trying to determine how best to represent the architecture and design of a software system is that there exist many distinct concerns. It is very hard to present all of them using a single diagram, and even if one succeeds, the usefulness of that diagram is doubtful. Thus it is common to separate these different concerns.

Kruchten's landmark paper on architectural views considers that there are "4+1" key architectural views [Krua]:

- Logical View — describes the conceptual decomposition of system into modules, layers, and other architectural constructs.
- Process View — describes the run time decomposition of system into processes and emphasizes concurrency and synchronization aspects.
- Physical View — describes the mapping of software components onto hardware components and emphasizes distributed aspects of the system.

- Development View — describes the source code and other development modules organization in a development environment.
- Use-Case View — this view unifies all the previously mentioned views through the use of use-cases.

For our purposes, we take a slightly different set of views:

- Problem Domain View (Analysis View) — represents the structure of the system that is supposed to be supported and/or automated using the software system.
- Logical View — represents the structure of the software as represented in the code, and different layers of abstraction built on top of it.
- Run-Time View — this is the unification of the previously mentioned process and physical view.
- Test View — represents testing framework.

Each of these views presents two kinds of system properties, static and dynamic. Notation used to capture these views should provide support for both properties. We will go into more details on architectural views when we present their use within the context of our research.

2.1.2 Reference Architecture

A reference architecture is the common architecture model for a family of products. It represents the structure that is commonly found, possibly with slight variations, in products that belong to that family. There are some well-known reference architectures, for example, for compilers [ASU85, SD96] and operating systems [SGG01]. The main purpose of creating a reference architecture is to codify knowledge about a particular product family, and to serve as a starting point in building a new product. It can also used as a guide to control the development and prevent anarchy in the system, usually by reengineering the system to fit its reference architecture more closely.

2.1.3 Architectural Refactorings and Repair

Refactorings are changes made to the code that improve its design in some way. Continual refactoring is an important feature of agile methodologies, as it helps to compensate for the lack of upfront design activities. A catalog of refactorings can be found at [REF], and a definitive guide to performing them is [Fow99].

Of particular interest to us are the refactorings that affect the architecture of the system. Some of them are presented in [TGLH00], together with a few successful applications. For example, architectural refactoring is a process of separating the user-interface functionality from the application logic in software systems where these two separate concerns were merged together.

The current problem with existing refactorings is that they are not automated and are tedious to perform manually. More robust support for automatic refactoring has to be developed and tightly integrated with the other development tools — one example of such a tool is refactoring support integrated in IntelliJ IDEA integrated development environment [IDE].

2.1.4 Architecture Reengineering and Visualization

One of the main uses of the software architecture is to help one understand the system. There have been many efforts recently to automate architectural reengineering and to present this information visually. Most of these tools have been used exclusively for the visualization of the static structure of the system and interconnections between its components. Examples of these kinds of systems include PBS [PBS], Rigi [Rig], and SHriMP [Shr].

The usefulness of such systems is in their ability to present us with important architectural issues, while hiding lower-level design decisions. For example, the PBS system allows us to analyze a software system at different levels of abstraction, with the lowest level being the source file level. It also allows us to analyze the dependencies between different subsystems, and the dependencies propagation through function calls, variable references, etc.

2.2 Quality Attribute Theory

Quality attributes are the descriptions and specifications of the characteristics of a software system and its ability to fulfill its intended functionality, while satisfying all the imposed constraints.

The study and use of quality attributes has made many contributions to software engineering practice. Goal-oriented requirement engineering processes have helped capture and model a wider range of requirements than previously possible, improve requirements traceability, and facilitate the process in general [MCY99]. Attribute-Based Architecture Styles (ABAS) have allowed qualitative reasoning about the use of a particular architectural style [KKB+99]. Architecture Tradeoff Analysis (ATA) method relies upon the use of quality attributes to analyze and express the architectural tradeoffs [KKC00]. Quality attributes are the initial artifact for several architecture design processes, including the Architecture-Based Design Method [BBC+00]. Besides providing the driving force, quality attributes also serve as the connection among all these techniques and methods [GY].

All of the previously mentioned techniques are based upon a solid understanding of the interdependencies among attributes, especially conflicting ones. Several studies have tackled the problems of complex dependencies between the attributes and how to manage them [BI96, vLL00].

Why do we need quality attributes? Building software right is hard and it requires much more then getting the functionality right. There are many constraints and properties that have to be satisfied. These include performance, security, availability, usability, modifiability, etc. All these properties are considered to be quality attributes, and are usually categorized according to when they are observable and measurable:

- runtime qualities — can be observed while the system executes, and
- non-runtime qualities — can not be observed and measured while the system executes.

Many of the system qualities depend on the architectural decisions. In fact, the main purpose of the software architecture is to satisfy certain system qualities. These include both, runtime (extensibility, modifiability, portability, etc.) and non-runtime qualities (security, scalability, reliability, etc.).

One of the important aspects to emphasize is that quality attributes are usually considered to be non-functional requirements. This is not necessarily true as many of them are functional requirements depending on the perspective. For example, security is often considered to be a non-functional requirement by the developers of the system, but for security specialists and implementors, security is a functional requirement achieved through the development of additional

functionality of the system.

The main drawback of all quality attributes is that they are not easily measurable. This makes them hard to specify in concrete terms. Because of this, there is a high level of ambiguity and subjectivity when dealing with quality attributes.

In order to optimize the qualities of the system, it is not enough to consider only the quality attributes and their priorities at one moment in time: The qualities of the system change, and the stakeholders' quality priorities change. We need a way to prioritize and estimate changes in quality attributes. The systematic study of these changes, and how and why they have occurred, helps us achieve this.

In this chapter, we have presented some of the basic concepts and techniques that we will see over and over in the rest of the dissertation. In the next chapter, we present the theoretical aspects of our software architecture recovery process.

Chapter 3

Agile Architecture Recovery Process

In this chapter, we present our architecture recovery process. In the first part of this chapter, we discuss architecture recovery issues and problems, including techniques used to solve them. In the second part, we present our architecture recovery process in a development process neutral way. In the third part, we discuss possible ways of adapting and using our process and techniques for the direct architecture recovery of a legacy application. Our process and techniques are applied in a context of direct architecture recovery of an application as presented in Chapter 4.

3.1 Architecture Recovery Issues and Goals

In order to understand and appreciate the value of architecture recovery, and to motivate its incorporation in a development process, we must first discuss some concrete benefits that knowledge about an existing architecture and its use in development bring to a project:

- Understandability — An architecture provides an abstracted high level view of the design of a system. This is of crucial importance in development of complex systems. Architectural views provide a way to present separate concerns in a manageable way, which leads to easier and faster understanding of a system and its interconnections.

- Modularity — A good architecture allows us to break down the system in smaller, less interdependent components, which can be developed separately. This increased control

of complexity makes it easier to manage larger number of components and to replace old components by newer better ones.

- Flexibility — Easier management and modification of components, run-time system modification and upgrading are all achieved through a good architecture.

- Cost Reduction — All of the previously mentioned benefits directly contribute to decreased cost of system development and maintenance.

As mentioned earlier, most software systems are built on top of a pre-existing code base, such as legacy applications and large libraries. Many of these applications are built without the careful use of forward design practices, and for most of them, their structure is not documented or the documents are obsolete. As qualities of our new application will depend largely on qualities of legacy systems used, we need a way of recovering and modifying the architecture of these legacy applications. Also, processes that do not emphasize forward design, and requirements that constantly change make it even harder to achieve major system goals.

The main goals of a successful architecture recovery process are to produce artifacts that describe:

1. the actual architecture of the system under consideration, and

2. the rationale of that architecture — why architecture is as it is.

The second goal is considerably harder to achieve since it includes recovery of forward design decisions made, external influences that produced them, and alternatives that were considered and why they were not implemented. The aim of our approach is to address both of these goals. In addition, in order to be an efficient and lightweight approach, our process aims to achieve the following goals and constraints:

- Minimal additional developer's training.

- Low risk incorporation in the development process.

- Robust roundtrip tool support.

- As much as possible based on "best practice" programming and design techniques.

- Minimize design activities and maximize programming — to be done by identifying crucial design and recovery activities, that stay stable so that the work invested in building them is profitable and that work performed on updating them is minimized.
- Minimal and simple set of artifacts directly usable later for forward engineering and study of evolution.
- Use of the forward engineering principles to recover architecture rationale.

3.2 Architecture Recovery Process

Ideally, the most influential force that drives architectural design should be the system's functional requirements and quality attributes. While often this is the main force that shapes the architecture of a system, there are many others: different technical, business, and people-oriented influences shape architecture in both positive and negative ways. For example, in a company that has development teams at different locations, actual work division can essentially dictate the architecture of a system. Another example is when management imposes the use of a particular technology, which is not an ideal one for the problem at hand (*e.g.,* use of object-oriented technologies for development of real-time systems for which the speed of execution is of crucial importance).

The fact that architecture is primarily derived from and influenced by quality attributes in the context of functional requirements imposes that an architecture recovery process should trace from the concrete architecture of the system back to the actual requirements that originally shaped it. At the same time, the analyst that is performing architecture recovery must isolate other influences and their effects on software architecture. Therefore, the main groups of activities performed during architecture recovery are:

- discovery of concrete architecture of the system;
- discovery of functional requirements and quality attributes as a major force behind architectural decisions;
- recovery of architecture design decisions that have led to actual concrete architecture of

the system, and identification of other possible architectural solutions and their advantages and drawbacks; and

- identification of other factors that have influenced the architecture.

The first activity is physical architecture recovery, and last three concern architecture rationale recovery, which together make the complete set of architectural artifacts.

As the first three activities and artifacts produced are of major importance for successful further development of the system, the main emphasis of our process is on them. The last activity is of secondary importance for the short term development of the system that usually follows recovery process activities, but of a major importance for long term development activities and improvement of business and development model.

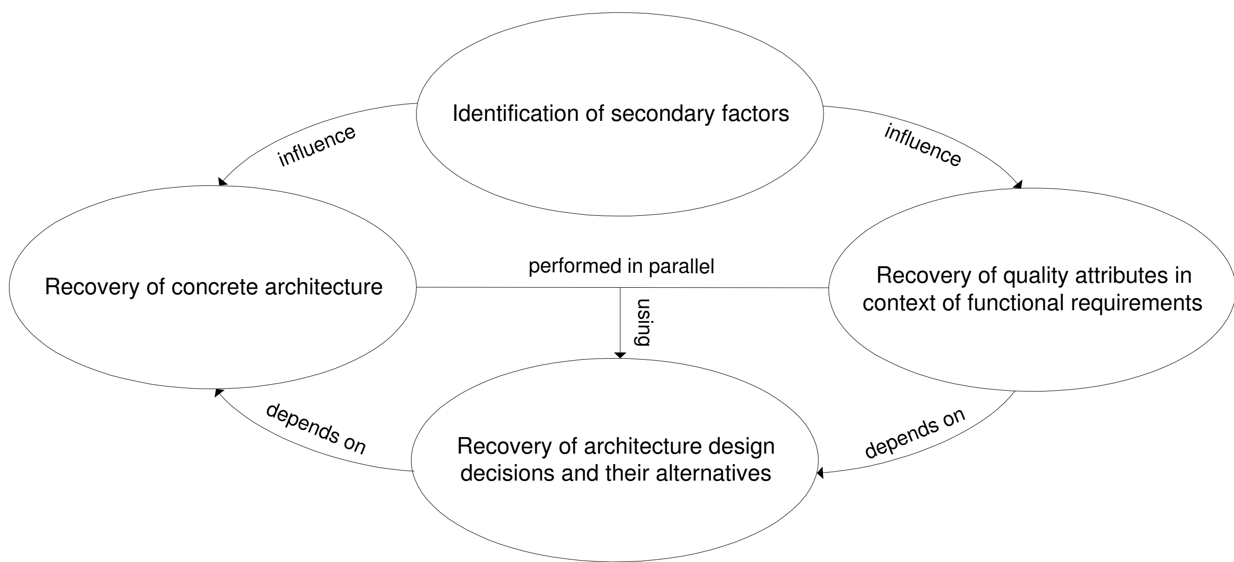

Figure 3.1: Major Architecture Recovery Activities

Figure 3.1 shows the main relationships among these activities. These recovery activities are largely performed in parallel, and depend on each other. This has led to an essentially iterative nature of our recovery process. We discuss this iterative nature later in this chapter when we introduce actual process steps.

We define our process in three steps. In the first one, we introduce major concepts that participate in the process, including sources of information, different domains of concern, different stakeholders, etc. We relate them using our architecture meta-model, which is used to keep our process focused and consistent. In the second part, we introduce the techniques used to manipulate these concepts, and architecture recovery artifacts that are produced as the result of application of these techniques. In the last part, we introduce the process steps that relate these different techniques, and provide guidelines on how to perform them.

3.2.1 Architecture Meta-model

In this section, we introduce the essential concepts that occur in an architecture recovery process. Architecture recovery spans several domains of concern, and an analyst has to capture all concepts and relations that occur in these domains. In order for a process to be focused, and architecture recovery efficient and useful, we have to choose a manageable subset of these concerns in a way that will maximize the benefits of capturing and documenting them.

The first major division of these concerns is on:

- business domain architecture, and
- computer system architecture.

Business Domain Architecture

A computer system is often a part of a larger business system, and serves as a resource to accomplish certain business goals. During an architecture recovery process, we need to study different aspects of the business domain in order to understand our software system. A large amount of information about business architecture is gathered, but often much of that information is lost and not documented during reverse engineering processes that do not value that type of information. Our recovery process tries to capture and preserve this information as it is very valuable for long term development goals, and is a major source of architectural influences.

There are four main sets of concepts that describe business architecture:

1. *Business Resources* — All entities, both physical and abstract, that exist inside a business environment. These include people, information, different computer systems, business

supplies and products, etc. These are entities that participate in business processes. A subset of these resources is a source of modelling concepts for software systems built using object-oriented or component-based methodologies. The value of tracking and preserving knowledge about these concepts is in the fact that they serve as a tool to perform analysis of our software system architecture, and to track changes to business and software system since when system was built as a part of evolution study and to evaluate how well a software system reflects today's business needs. A computer system for which we are performing architecture recovery is a resource within one or more business ecosystems.

2. *Business Goals* — The purpose of performing a business activity is to achieve certain goals. Goals can be decomposed into subgoals, and at a certain level of decomposition, we reach business goals that have to be satisfied directly by our software system and its architecture. The study of business goals allows us to evaluate how well our software system's goals conform to them, and how we should improve our system. An important part of this work is the study of the evolution of these goals so the architecture of our system will be able to support future goals and to remove obsolete ones.

3. *Business Processes* — A system for which we are recovering architecture may participate within several business processes in order to help achieve certain business goals. The most common forward software analysis technique is discovery of use-cases to capture requirements. These use-cases describe sub-processes of larger business processes that are automated by our system. It is important to understand business processes as they relate all the use-cases of our system, which in turn relate requirements that our system has to satisfy. This allows us to study the architecture of a system within a context and to analyze the architecture.

4. *Business Rules* — Business rules are a major source of constraints on software system. Many of these constraints directly influence software architecture. As such, it is important to understand them and keep track of them, for example, to remove architectural limitations imposed by constraints that do not hold any more.

We will discuss how these concepts relate to each other after we introduce computer system architecture and its concerns. We discuss how to capture and preserve business architecture when we introduce techniques used in our process.

Computer System Architecture

Many software development technologies, such as object-oriented development technology, allow us to simulate concepts that exist in a problem domain. For our purposes, we define the following architecture concepts that we will keep track of:

- system,
- subsystems,
- modules,
- connectors,
- processes,
- logical processes, and
- hardware devices.

System concept defines the outermost boundary of the software system under consideration. Same as for all other concepts, the amount of information and its scope depends on the architecture view in which it is used. It serves as a container for all other concepts, and defines the computer system as a resource in the business model.

Subsystems are abstract concepts that serve as abstraction tools for management and abstraction of actual physical modules, connectors, and processes. They serve as containers and building blocks of the whole system. Depending on the architecture view, they capture different concerns of the system.

Modules are basic architectural building blocks. For example, in the logical view they represent concepts that occur in business domain, and in the implementation view they represent code units. Modules are abstractions of basic building blocks of the system, depending on the development technology used. For example, if a procedural paradigm is used, they are sets of logically related procedures and data (usually ones that exist in the same source file), in component-based paradigm they are components, and in object oriented they are groupings of one or more classes.

Connectors are abstractions of communication mechanisms and channels that exist in a system. Their size and complexity vary from simple procedure calls to connectors built from several modules and hardware devices.

Processes are physical, run-time processes that perform activities that fulfill goals of logical processes. They are allocated to possibly many different processing nodes, and are basic building blocks of run-time view.

Logical processes are white-box use-cases. The difference between them and regular, black-box, use-cases is that they include descriptions of which activities are performed within the system. We will discuss them in more detail when we introduce architectural use-cases in architecture recovery techniques section of this chapter.

Hardware devices are concepts that occur in run-view, and represent actual hardware device that are parts of a computer system.

Figure 3.2 shows the main concepts, and main relationships at the architectural level. This is not the only possible decomposition, but is a useful, minimal one that will help us focus on the main architecture issues without getting lost in many details, which are usually not necessary. Simplicity of the decomposition is also in the spirit of our agile approach, and allows us to abstract from the details of used low-level development technology.

Now we will present techniques that are used to recover and present these concepts in order to provide a useful architectural description.

3.2.2 Architecture Recovery Techniques

Before we embark into the discussion of particular architecture recovery techniques and artifacts, we summarize the main goals of a successful recovery. These goals will be used to classify and relate specific techniques. These goals are:

- Discovery of the current structure of the system.
- Discovery of the main influences that have led to that architecture — most important are quality attributes (in context of functional requirements).
- Discovery of the design decisions that have led to the current architecture of the system and possible alternatives.

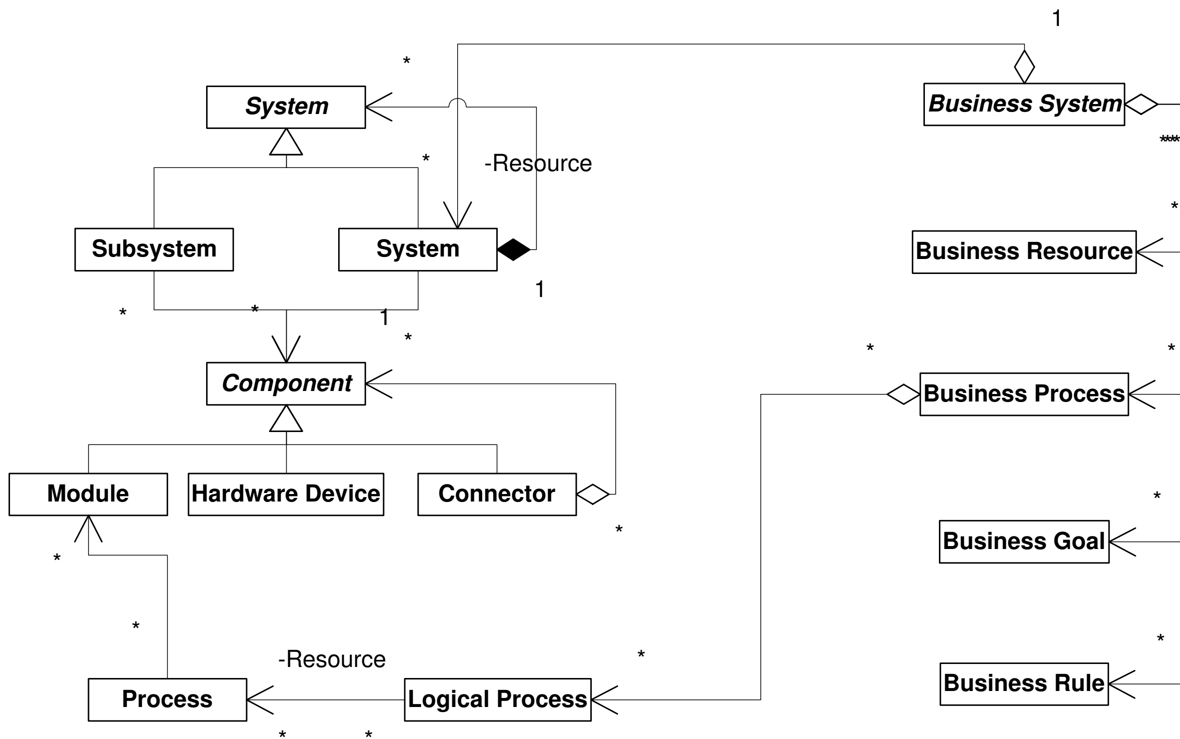

Figure 3.2: Architecture Meta-model

Discovery of Present Architecture

Recovery of current architecture of a system is a goal into which a lot of research efforts have been invested. There exist many reverse engineering tools that can help, business reverse engineering processes and techniques, formal methods, etc. Nevertheless it is a very difficult and demanding task to perform.

Our approach to the recovery is iterative and view-driven. Architectural views provide separate conceptual approaches that allow us to focus on recovery of only a portion of architectural concerns and facts at a time. Iterative means that we do not focus on completion of architecture recovery from only one perspective (view) at a time, but iterate among recovery from different perspectives and use feedback from one to improve another.

The main work flow is:

1. Initial recovery from problem domain perspective.

2. Recovery from logical, run-time, and test perspective.

3. Recovery from problem domain perspective, and iteration to step 2.

One of the main goals of a good architecture is to preserve problem domain concepts in software architecture. Therefore, during recovery we pay special attention to discovery of actual mappings from problem domain view to other views, and vice-versa. This is done by tracking which concepts from one view allow discovery of concepts in another. Only architecture-relevant concept dependencies are recorded.

Problem Domain View

In order to perform architecture recovery from a problem domain perspective, we use forward requirements and business engineering techniques such as employee interviews. The focus is on discovery of business resources, goals, processes, and rules that are closely related to our software system.

Two possible approaches that we are using, depending on the type of system under consideration, are actor-driven and process-driven approaches. The actor-driven approach is useful for systems where actors, as active resources, control processes that occur inside the system. For example, in business environments, employees as active resources control most of the business

processes, so for recovery of information systems from a problem domain view perspective, we should use an actor-driven approach. The main steps are:

1. Discover an actor.
2. Discover the goals of that actor.
3. Discover the processes performed to achieve these goals and the resources used or affected by them.

On the other hand, the process-driven approach guides recovery more effectively in business domains where actors are of secondary importance. For example, for an industrial control system problem domain, the main focus is on processes that occur within it and actors are of secondary importance. The main steps are:

1. Discover a process and the resources used.
2. Discover which goals are achieved by that process.
3. Abstract goals at the level of business goals.
4. Identify which resources are actors.

Contrary to business engineering needs, for our purposes we do not require a large amount of detail. The main artifacts produced are:

- Actor-Goal List
- Goal-Process List
- Process-Resource List
- Business Rules List

The main purpose of these artifacts is to preserve knowledge about identified concepts and relationships among them. We use a simple list of items although other formats, such as UML diagrams, are possible. The main reason for using simple lists is to minimize the effort in producing and updating these artifacts. We present a sample artifact in Chapter 4 in the context of our case study.

Logical, Run-Time, and Test Perspective

The most important perspective for our purposes is the logical view of the system. It serves as a bridge between the problem domain perspective and the implementation perspective. It also has a strong influence on the run-time and test aspects of the system.

Logical view is composed of a set of concepts, some of which appear in the pure problem domain, and some in a pure implementation view. The main value of this view lies in the fact that it provides a focus and is the main source of information for actual development of the software system.

This view is further subdivided in two different aspects, which occur naturally during the recovery process:

1. conceptual architecture, and
2. concrete architecture.

Conceptual architecture is a very high level view of a system's architecture, which occurs often as a transition step when one analyzes first architecture from a problem domain perspective and then moves toward analysis at the implementation level. Concepts that appear in artifacts presented at the conceptual architecture level are largely a subset of concepts that occur in the problem domain view.

On the other hand, concrete architecture is a more detailed view of the system and appears when one is analyzing a system using the bottom-up approach (*i.e.,* starting from source code and then abstracting concepts and mapping them to the concepts that occur in the problem-domain view). Concepts that appear in artifacts at this abstraction level are largely a subset of concepts from pure implementation view.

Our approach is mixed and encourages iteration in the refinement of artifacts at both abstraction levels. As conceptual architecture is closer to problem domain concerns, it is recovered using familiar object-oriented, or component-based analysis techniques and general problem domain knowledge. Concrete architecture is, on the other hand, recovered using reverse engineering tools like source code browsers. Finally, a mapping between these two abstraction levels are established, with special emphasis on mismatches, which are potential candidates for architectural refactorings.

Concepts that appear in this view are:

- system,
- subsystems,
- modules,
- connectors, and
- logical processes.

The main techniques used to produce artifacts in this view are:

- reference architecture, and
- responsibility based dynamic architecture representation (architectural use-cases).

The former was discussed in Chapter 2, so we discuss only the latter one here, and relate them in a section that describes how all recovery steps are performed together.

Use-cases — that is, narrative descriptions of domain processes — appear in different forms in all the phases of a development cycle. They are typically used as the artifacts around which development cycles are organized. When used this way, all the other activities and artifacts depend on them.

Use-cases describe the interactions between actors and the system. A use-case encapsulates responsibilities that are performed during a process by actors and by the system; that is, they indirectly describe responsibilities.

Architectural use-cases are use-cases that describe *logical processes* within the system. In our approach, these use-cases are not created by developers, but are generated using high-level responsibilities that are written as the part of the code documentation. The purpose of this generation is to document dynamic processes within the system. This is a technique used as the part of the logical view in order to present dynamic interactions in a comprehensible format.

Therefore, to present dynamic aspects of the system we use architectural use-case. For static aspects, many different notations can be used. The only requirement is that previously mentioned concepts (system, modules, connectors, and subsystems) can be clearly presented. Our tool is PBS (Figure 3.3), which provides a simple box-and-arrow notation to capture these concepts and relations among them; however, any UML CASE tool can be used to present these concepts.

We will present sample architectural use-cases, static diagrams, and provide guidelines on their construction in Chapter 4.

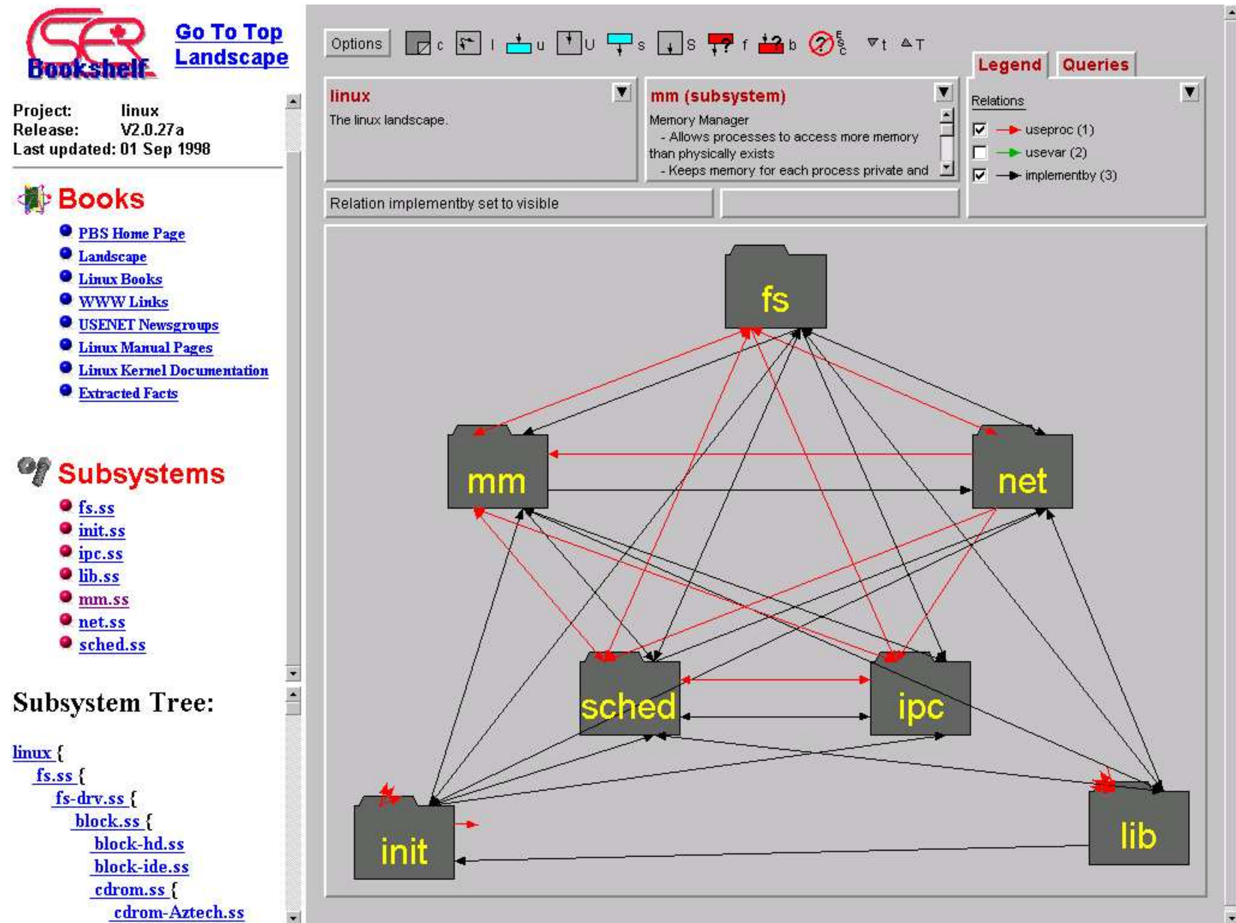

Figure 3.3: PBS: Architecture of Linux Operating System

The Run-Time view in our process is limited to only a basic description of computer system hardware and process distribution. Only main processes are discovered, presented on a diagram, and mapped to their processing nodes. The only purpose of this diagram is to allow us to map the main sets of functionality of our system to distributed aspects of business domain. We do not study it in more detail as the run-time view depends highly on the choice of hardware and middleware technologies, and the whole purpose of architectural recovery is to abstract above

these low-level concerns. More detailed study of run-time aspects of a system is appropriate during low-level design recovery of a system.

Test view is an optional view. It is mainly used when a system has an existing, extensive set of automated, software-based testing artifacts such as unit tests. Techniques used to recover testing aspects of a system are the same as the ones used for recovery from logical perspective, so we will not discuss them again. The main use of test view is in conjunction with logical view to provide a more complete mapping to problem domain view. The test view is especially important when studying the evolution of a software system, as it represents current external needs from business perspective (*i.e.,* requirements that our system as a resource is supposed to fulfill).

Now, when we know the state of the current architecture of the system, we need to find its rationale. The following two sections elaborate on this issue.

Quality Attributes and Design Decisions Recovery

Similar to recovery of architecture from the logical view perspective, the discovery of quality attributes is a two way process. In one direction, quality attribute recovery is a requirements engineering activity where one analyzes a business system to find out which qualities a computer system as a business resource has to have and fulfill. In the other direction, quality attribute recovery is a technique performed in conjunction with design decision recovery, and basically represents an answer to a question of why a particular design decision has been made.

Quality attribute discovery from requirements perspective is a well studied activity, described in many requirements engineering texts [RR00]. Compared to the actual requirements engineering process, ours is simplified since we concentrate only on discovery of architecture-relevant quality attributes. As a main guideline, we have an initial set of commonly occurring quality attributes for a particular problem domain (there are many classifications of quality attributes) such as reliability, performance, etc. For each one of these high-level quality attributes we are discovering, analyzing, and documenting *quality facts* — simple statements that quantify these quality attributes in the context of functional requirements. For example, “in a case of main system failure, the monitoring system must restart the main system within 10 seconds”. These quality facts are documented in *quality attributes table*, and are related directly to high-level quality attributes, and functional requirements within which they are analyzed.

Quality attribute recovery from the other perspective is guided by the analyst’s expertise in

forward architecture design techniques. Many architecture styles today are directly connected to a set of attributes for which they are particularly well suited [KKB+99]. The analyst's role is to discover architectural styles in the system under consideration and to deduce which attributes have led to it. Also, for a given set of attributes, analysts must consider which other architectural styles might be appropriate, and to evaluate their advantages and disadvantages compared to the set of styles already used for the system under consideration.

Attributes that are discovered as a result of the design decisions recovery process, and that do not map to the actual needs of the business system, are isolated and emphasized. This is done since architectural styles that traced back to these attributes exist because:

- there is some other architectural influence, such as division of work among developers, or
- these quality attributes have existed before at some point in time, but do not exist any more, and that architecture style is probably a candidate for architectural refactorings.

To record this architectural rationale, we use two documents:

1. a document that relates quality facts through design decisions to the actual solution, which is ideally expressed as a set of architectural styles, and
2. a document that contains alternatives to the existing particular solution.

The first document also includes other discovered, non-attribute, architecture influences and their results in the architecture. Even though this information can be presented in several different formats, we use tables, as they are simplest to build and maintain. Another obvious possibility is to use diagrams, but we do not use them due to the very large amount of information that would add too much noise to the diagram. Also, diagrams are harder to update and maintain. We will present examples of these tables in Chapter 4.

Now when we have defined our set of activities and artifacts, we will discuss how they fit together for a direct recovery of a legacy application.

3.3 Architecture Recovery Process Steps and Legacy Systems

Although we have limited our recovery process to only a small subset of activities and artifacts produced in order to achieve our goals, Figure 3.4 shows that there exists still a relatively high

complexity due to the many relationships among these activities and processes. Rectangles represent artifacts that are of major concern to recovery, and ellipses represent artifacts that are either transitional or optional artifacts. Arrows represent flow of activities and information. From the first look at this diagram, we can see that there is a lot of iteration within the process. A major challenge is thus to order these steps in a such way that they can integrate seamlessly with our current process and produce optimal results.

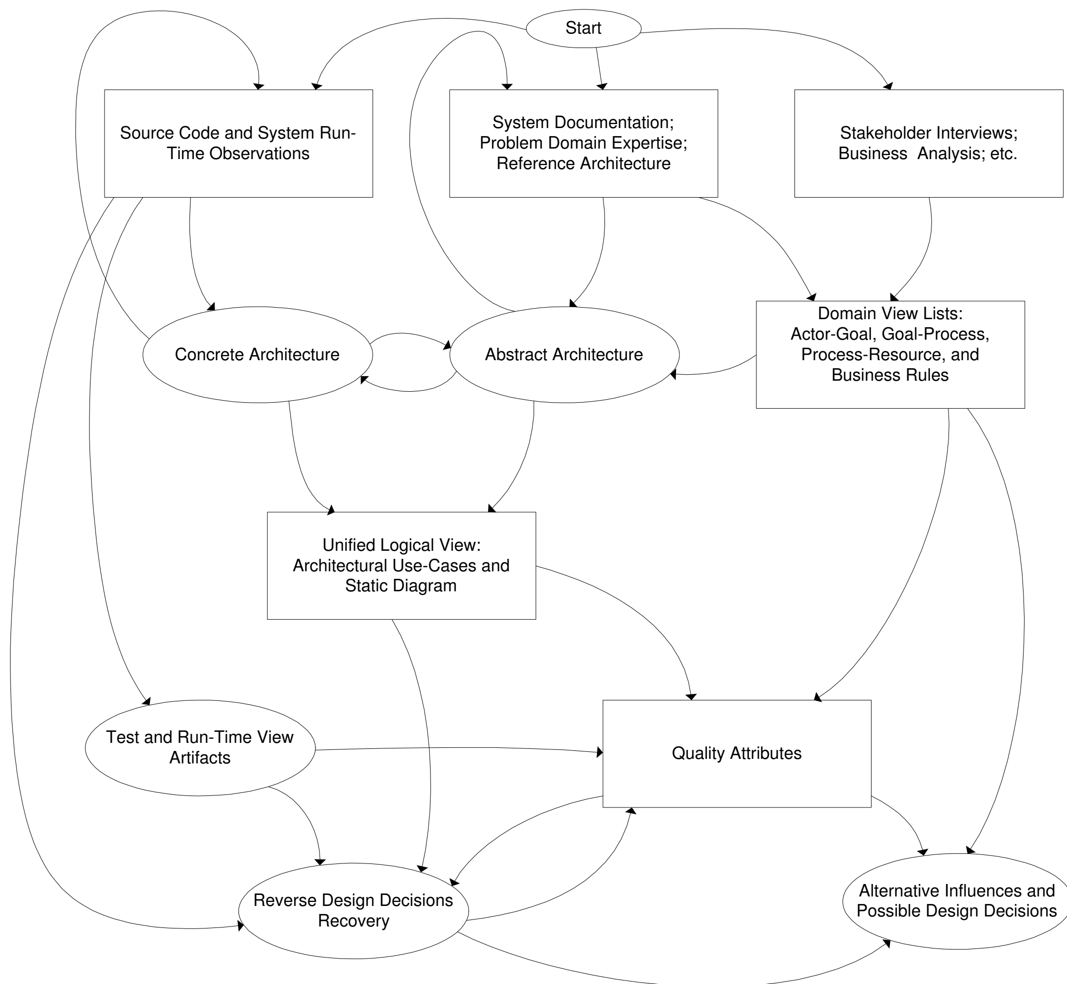

Figure 3.4: Architecture Recovery Artifacts and Relationships

A very common task is a one-shot architecture recovery of a legacy system. It is usually performed in isolation from the actual development process. Its advantage is that several analysts can focus only on the architecture recovery of the system, without having to worry about changes to the system during recovery, other development activities, and so on. This makes it simpler to organize and perform all the steps of the recovery process.

This ability to focus only on recovery allows us to perform major steps in a waterfall-like fashion. These are the highest level steps that are performed sequentially:

1. project elaboration,
2. architecture recovery, and
3. architecture presentation.

Project Elaboration

Software architecture recovery, as any other software engineering project, requires time, resources, and involves a certain amount of risk. Software architecture project elaboration phase is usually a subcomponent of a development process elaboration phase, during which stakeholders are supposed to evaluate the feasibility and usefulness of architecture recovery of a legacy system in order to incorporate it into further development of a new system. Questions such as if it is less expensive to recover architecture of this system and improve it or to build a new system should be answered during this elaboration phase.

To be able to estimate the feasibility of the architecture recovery process, we must consider several key issues:

- Amount and quality of pre-existing documentation — especially important are relatively up-to-date functional requirements lists, initial goals of the system, user manuals, and deployment descriptions.
- Low-level quality of code — is the code self documented, what methodology was used to develop it, etc.
- Size of the system.
- Availability of initial developers and designers of the system, and level of current knowledge about the design of the system.

- Estimation of business changes and how they were reflected in the system — it is possible that a system is out of date and even if architecture is recovered, there will be a need for a complete refactoring of the system, which is often more tedious to do than to build a new system.
- Analysts' business and problem domain knowledge.

Other aspects that should be evaluated are the availability of general reverse engineering and source browsing tools, requirement gathering tools, and appropriate drawing tools.

If all these previously mentioned aspects are positive, one can relatively safely embark into a recovery process. Before the actual recovery is started, one should prepare existing documents and tools for recovery. This depends on the amount of information and tools available, but at minimum, one should have:

- Known functional requirements listed and grouped in related-functionality groups.
- A source code browsing and editing tool.
- A drawing tool.
- A text processing tool.

Now when a decision to continue the process is made, and initial set of artifacts and tools are ready, analysts can start the actual recovery process.

Architecture Recovery

In order to allow work distribution and separation of concerns, our recovery process is divided into the following steps:

Abstract architecture recovery, business domain analysis, and run-time aspects analysis — Using requirements engineering techniques, existing documentation, and problem domain expertise, analysts try to produce separately an initial abstract architecture of the system and domain-view lists. One of the main techniques that helps recovery of domain and conceptual architecture is actual use of the system, which aid in creating a high-level run-time architecture. It is important to note that much of the effort invested in early iterations is not on getting these artifacts perfectly right but on providing artifacts that will allow us

to perform following steps. After every iteration, set of artifacts produced is refined and improved.

Concrete and test architecture — Analysts try to map abstract architecture concepts onto source code and to discover mismatches. Responsibilities and invariants source code instrumentation is performed at the same time. These are used to make a summary of dynamic aspects of concrete architecture, and initial diagram of static building blocks is produced. Again, this is done with assumption that these will be refined in following iterations. At the same time, optional test view artifacts are constructed using the same techniques. Iteration is finished with unification of concrete and abstract architecture artifacts to reflect current full understanding of logical architecture of system, and to serve as input artifacts for the following iteration.

Quality attributes and design decisions recovery — While previous steps were mostly routine observation and analysis, this step depends highly on the analysts' software design and development expertise. Also, it depends on availability of codified architectural patterns and attribute quality knowledge for that particular domain. As a guideline, we will present a minimal set of these as a part of our case study. These two activities are very tightly coupled and the iteration between them is so high that they can be effectively considered as one, unified activity. Artifacts produced are quality facts list related to a set of design decisions, and a list of mismatches that indicate possible external architectural influences.

Alternative influences and possible design decisions Using artifacts produced in the previous step and domain-view lists, analysts try to discover possible external architectural influences, and possible design alternatives. The amount of effort invested in this step has to be critically estimated and justified. For example, if no changes to the system are supposed to be made, only very obvious alternatives should be documented. On the other hand, if a need for extensive refactoring is already observed, or if large drifts in the business domain are expected, analysts should try to identify as many external factors and alternative design decisions as possible. Activities from this step are strongly emphasized in software evolution recovery process, so if architecture is recovered as a part of evolution study, emphasis should be put on this step.

These steps form a recovery iteration, and a whole process consists of several iterations. Each subsequent iteration is more specific and detailed than the previous. A common way is to organize iterations following top-down architectural decomposition of the system. For example, first one is concerned with architecture at the system level, and next one at the level of one of subsystems.

Architecture Presentation

Depending on needs, during this step previously produced artifacts can be presented using different formats for different purposes. For example, if artifacts are supposed to be used for general understanding of the system by new developers, additional diagrams can be created that present information from tables in clearer way. Again, effort invested in creation of additional documentation has to be critically evaluated against cost criteria.

3.4 Summary

In this chapter, we have presented the overall organization of our process and high-level artifacts. We have intentionally omitted presenting the low-level steps and providing the definitive guidelines on how each step has to be performed. This was done in order to keep the process adaptable, and to stress the importance of agility and use of the different techniques to achieve the same goals. For example, we did not enforce the use of any particular tool and its specific recovery steps for the recovery of the concrete architecture. Nevertheless, in the next chapter we tackle these low-level process steps and issues (including recommended guidelines, minimal sets of crucial design techniques, etc). The next chapter describes a case study, which was used to validate the applicability and effectiveness of our process.

Chapter 4

Case Study: X MultiMedia System

In this chapter we present an architecture and evolution recovery case study of the X MultiMedia System (XMMS) [XMMa]. The main purpose of this case study is to evaluate our process by applying it to the recovery of a real world system. The main factors that have led to the choice of XMMS as a guinea pig are:

- It is a real, industrial strength, multimedia application in very wide use.
- It is of non-trivial size, 65,000 lines of code, but it is also small enough to be tractable for our purposes.
- It is in active development, and multiple versions of its source code can be obtained.
- It is developed using an open source licence, which means that we can examine it and publish our results freely.

We note that while 65,000 lines of code is not usually considered "large" by industrial standards, it is often the case that architecture recovery of large systems often concentrates on major subsystems of similar size to XMMS system as a whole.

In the first part of this chapter we introduce XMMS, including its description, development team, and history. In the second part, we present architecture recovery. In the third part, we present a focused evolution recovery of the system.

4.1 XMMS: Introduction

The development of the player started in late 1997, and since then it has become a very popular and widely used multimedia application, particularly in the Linux community. It runs on many Unix based operating systems and a goal of the XMMS is to have all functionality that WinAmp [Win] player has, plus some additional features available only on Unix.

XMMS has a rich set of features. Beside the basic features, such as audio files playing and management, XMMS provides many additional features that provide rich user experience, such as support for different visualization skins, sound enhancing plugins, etc. Many of the additional features are available as the plugin components that are developed by third parties, and whose development is possible due to an open plugin architecture of the system.

The development team consists of three main developers. The core system programming is done by Peter Alm (peter@xmms.org), the graphical interface is done by Thomas Nilsson (thomas@xmms.org), and everything else is done by Olle Hallnas (olle@xmms.org).

4.2 XMMS: Architecture Recovery

This section describes architecture recovery as it was actually performed. Architecture recovery consisted of two iterations, second one building on the first one, refining and improving recovery artifacts. This section is organized accordingly, starting with project elaboration and then describing both iterations.

4.2.1 Project Elaboration

As this is a moderate size recovery project, an extensive project elaboration is not of crucial importance. Nevertheless we present some major issues.

Following are sample artifacts that allow us to estimate feasibility of the project:

- Amount and quality of already available documentation — Documents that are available are a summary of high level features of the system [XMMb], and user manual [XMMc].
- Quality of code — System is developed using C language. A quick overview of code showed a good coding style, and relatively low amount of comments.

- Size of the system — Table 4.1 summarizes some basic system metrics. They are produced using a source code analysis tool Understand for C++ [UND]. Interesting to note is the increase in the size of the system from the first to the last considered release was 36% in the number of functions and 38% in the number of lines of code. The greatest increase in the size was in the release 0.9.5, which was approximately 15.3% in the number of functions and 12.4% in the number of LOC. Also, we observed a stable percentage of the comments, 12-15%, but most of them were different copyright rules and disclaimers, and not the actual program comments.

- Availability of initial developers and designers — None of developers is available for consultation. Also, no architecture documentation was available.

- Analyst's business and problem domain knowledge — No previous knowledge about architecture of multimedia applications. Familiarity with systems Windows platform counterpart WinAmp [Win] from user perspective.

	Files	Functions	Lines	Lines Code	% Comment
XMMS 0.9.0	154	948	38101	29844	12
XMMS 0.9.1	154	950	39103	31061	11
XMMS 0.9.5	173	1122	45503	35466	13
XMMS 0.9.5.1	175	1129	45829	35710	13
XMMS 1.0.0	192	1230	50379	38900	14
XMMS 1.0.1	192	1233	50631	39107	14
XMMS 1.2.0	212	1287	56217	43126	14
XMMS 1.2.1	212	1286	56228	43124	14
XMMS 1.2.2	214	1290	56337	43185	15
XMMS 1.2.3	216	1322	57020	43599	15
XMMS 1.2.4	217	1345	58115	44446	15
XMMS 1.2.5	232	1482	63124	48125	15

Table 4.1: XMMS: Basic System Metrics

Detailed risk analysis of these and similar factors is out of scope of this thesis, and more appropriate for larger systems. Other aspects that should be evaluated are the availability of general reverse engineering and source browsing tools, requirement gathering tools, and drawing tools. If all these previously mentioned aspects are positive, one can relatively safely embark into a recovery process.

For our project, the following documentation and tools are available:

- High-level feature list and user manual [XMMb, XMMc].
- A source code browsing and editing tool: Understand for C++ [UND].
- A drawing tool.
- A text processing tool.
- Concrete architecture visualisation system: PBS [PBS].

When the decision to proceed with the project is made, and an initial set of tools and artifacts prepared, one can start the actual architecture recovery.

4.2.2 Architecture Recovery: First Iteration

Although almost all aspects of architecture recovery are tackled during each iteration, the emphasis is put more on some of these aspects than on others. During the first iteration, most of the emphasis is put on recovery of conceptual and domain aspects of architecture. Run-time architecture has a lower emphasis, and concrete and rationale are tackled the least. The reason for this is in the natural work-flow and availability of input for each aspect recovery. Also, information from one aspect is used to improve others.

During the first iteration, emphasis is on recovery of high-level details. These details are subsequently refined in following iterations, which continues as long as we achieve desired level of architectural detail for our purposes.

The main reason for this iterative, top-down recovery is a need to efficiently conquer this large amount of information. Also, it is more efficient to recover details from a particular perspective if there exists feedback from others.

The first step is to distill available documentation for potentially architecturally relevant features and concepts (this is possibly done at the end of the project elaboration phase). We extracted this information from high-level feature list and user manual [XMMb, XMMc]. These features and concepts are sorted into related categories. Table 4.2 presents core features of the system; Table 4.3 shows user interface and visualisation features; Table 4.4 contains features in form of plugins; and Table 4.5 represents other features.

Files search
Volume control
Balance control
Shuffle play
Repeat play
Equalizer
Playlist editor

Table 4.2: General System Features

Spectrum analyzer
Oscilloscope
One line mode
Timer elapsed
Timer remaining
Double size option
Winamp 2.0 user interface skin support
Gnome/Afterstep/WindowMaker window docking
GTK Interface for requesters (with theme support)
Auto remove borders if the windows manager has support for it

Table 4.3: User Interface and Visualization Features

Visualization — control different visual effect
Effects — control and alter sound in differen ways
Input — processing different streams and formats
Output — controls output stream and formats

Table 4.4: Plugins Features

Streaming/Shoutcast(1.0/1.1)/Icecast support
Fast jump in playlist
Scroll wheel support
Saving to wav file format option
Saving http streams to hard disk
HTTP authentication
Playing mpeg layer 1/2/3
Playing wav and formats supported by mikmod
Proxy authentication support
Support for other systems (FreeBSD, Solaris, LinuxPPC, AIX, Irix)

Table 4.5: Other Features

Domain, Conceptual, and Run-Time Architecture

As we have mentioned earlier, the run-time architecture presentation in our approach is a simple, high-level description of the system's deployment and distribution properties. As our system is just a stand-alone, desktop application, we will not spend much more time describing it. Nevertheless, experience in using of the system provides us with an opportunity for concept and feature discovery. This is especially important since our application is event-driven. For such systems, an actor is a main driver and controlling force, and the user interface provides an excellent source of information about the features of the system. This leads us into use of actor-driven recovery approach for domain architecture (Section 3.2.2).

Domain View. As with all artifacts in early iterations, our goal is not to discover all actors, goals, and processes at once. The aim should be to discover as many as possible, but not spending a large amount of time searching for hidden processes, and perfect relationships among all these concepts. Many of these hidden actors, goals, and processes will become evident during recovery of architecture from other perspectives. These concepts that are discovered later will be analyzed in more detail as a part of domain view recovery in following iterations.

The main stakeholders that are discovered during the first iteration are end-users, developers, multimedia providers, and the open-source community. As end-users are the only ones that conform to the definition of actors [FS99], they are shown in Table 4.6 together with their main goals. If architecture recovery is used to recover and improve the business model, other stakeholders can also be included in the table and subsequent analysis. Since our goal in this case-study is recovery of the architecture for software system understanding and improvement, we will not further consider these additional stakeholders.

End-user	Play multimedia content Record multimedia content Store multimedia content

Table 4.6: Iteration 1: Actor-Goal List

Goals are achieved through processes, and processes use certain resources. Processes have to conform to certain business rules. Tables 4.7, 4.8, and 4.9 summarize relationships among these concepts.

Play multimedia content	Obtain multimedia content Setup multimedia system Adjust listening/watching preferences
Record multimedia content	Obtain recording hardware Setup multimedia system Adjust recording preferences
Store multimedia content	Obtain storage facilities Produce required multimedia file format

Table 4.7: Iteration 1: Goal-Process List

Obtain multimedia content	Multimedia CD/DVD Internet access
Setup multimedia system	Multimedia software Computer system including monitor and speakers
Obtain storage facilities	Re/Writable CD/DVD Hard disks

Table 4.8: Iteration 1: Process-Resource List

Obey copyright rules

Table 4.9: Iteration 1: Business Rules List

Although these are simple and high-level lists, they summarize the main concepts that occur in the problem domain, and together with initial feature and concepts lists obtained from existing documentation, they provide guidance and a focus for the initial steps of conceptual architecture recovery.

Conceptual architecture. In order to recover logical view artifacts and to perform rationale recovery and analysis, one must be familiar with basic forward design techniques and patterns. For the following discussion, we assume the reader is familiar with GRASP patterns [Lar01] and patterns from [GHJV96, BMR+95].

The first step in conceptual architecture recovery is to build a static conceptual architecture presentation. The ideal way for recovering conceptual architecture is to interview the system's architects and developers. This way architecture recovery is a straightforward analysis and documentation of obtained results. As we did not have access either to developers or to any architectural documentation, we had to rely upon other techniques.

Techniques that we used to derive the initial conceptual architecture were:

- General reference architecture for stand-alone, GUI, desktop application — layered architectural pattern [BMR+95].
- GRASP patterns [Lar01] — they represent basic knowledge about the responsibility assignment rules and heuristics. These include low coupling principle, high cohesion principle, indirection to achieve separation of concerns, controller pattern, etc.
- Model-View-Controller pattern [GHJV96].
- Identification of separate architectural entities from initial feature and concept list.
- Domain view artifacts analysis for unification of feature, processes, and resources.

Figure 4.1 presents the static conceptual architecture that we derived. Arrows represent connectors which are in this case ordinary procedure calls and direct data access. The goal of this initial architecture was to allow us to document initial dynamic conceptual architecture.

Activities performed for initial dynamic conceptual architecture recovery are responsibility assignment of early discovered features to entities in recovered static conceptual architecture, and identification of major architectural use-cases using domain processes and feature list. Architectural use-cases are not fully developed and written down in this early stage since they

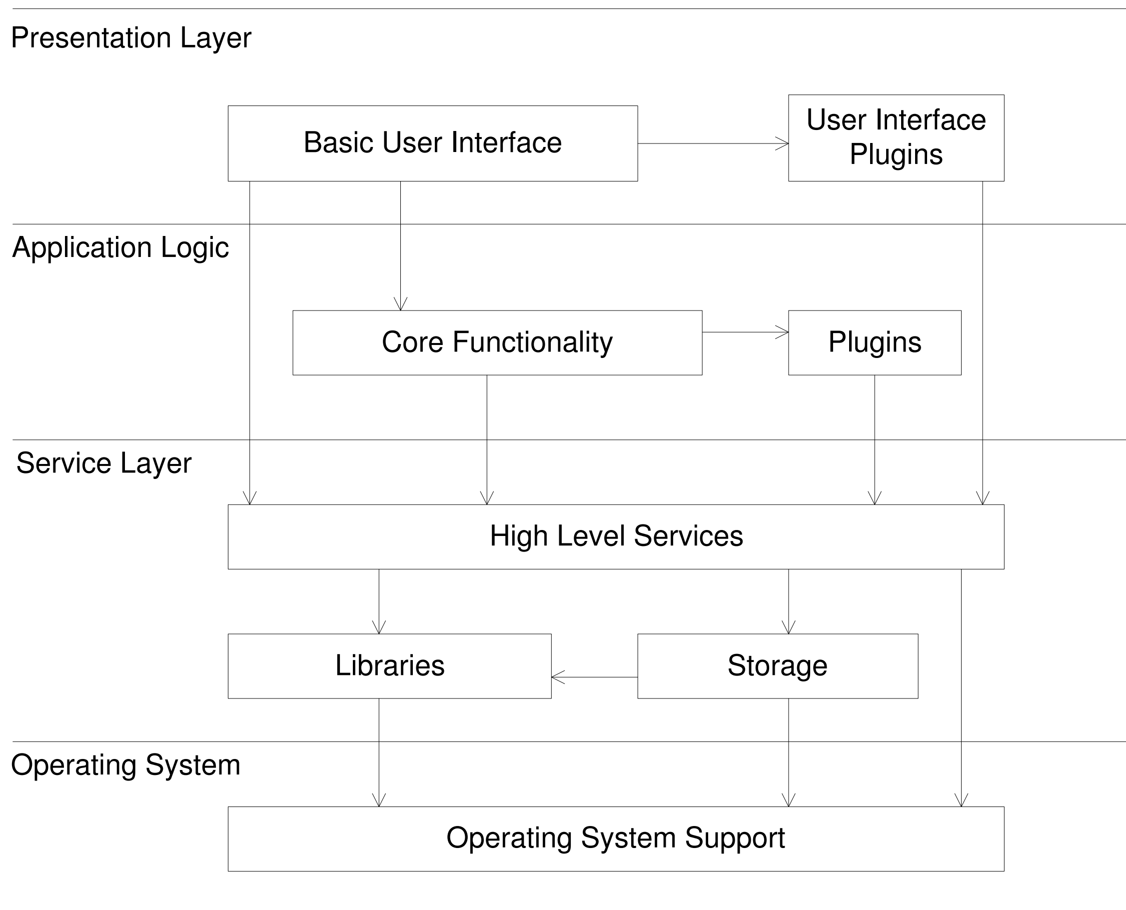

Figure 4.1: Iteration 1: Static Conceptual Architecture

would drastically change after concrete architecture recovery, and unification of the logical view. The main purpose of dynamic conceptual architecture at this stage is to serve as input and driver for concrete dynamic architecture recovery. Full use-cases are developed and documented in the next iteration. Tables 4.10, 4.11, and 4.12 summarize the subsystems' responsibilities.

Subsystem	Responsibility
Basic User Interface	Provide support for visualization plugins Manage spectrum analyzer Manage oscilloscope WinAmp skin support Support different sizes and full screen mode Display controllers Playlist editor Auto remove borders if the WM has support for it One line mode
User-Interface Plugins	Control different visual effects

Table 4.10: Iteration 1: Presentation(UI) Layer Subsystem Responsibilities

Now when we have an initial conceptual architecture, using it we will try to discover concrete architecture at the same level of abstraction.

Concrete Architecture and Rationale Recovery

While conceptual architecture tends more towards encapsulation of pure logical view aspects, concrete architecture tends towards capture of pure development view aspects. The main approach to concrete architecture recovery is study of source code organization. This step involves study of source code directory organization, namespaces (packages), code browsing, configuration files, and other source code artifacts. Code browsing is performed with the intention of discovering high-level features, and not performing detailed analysis of control flow, reference detections, etc.

Techniques that we used for this iteration were the study of source code directory structure and source code documentation. The major subsystems and their main responsibilities are:

Subsystem	Responsibility
Core Functionality	Seeking in files
	Volume control
	Balance control
	Shuffle play
	Repeat play
	Equalizer
	Track playing time
	Fast jump in playlist
Plugins	Streaming/Shoutcast(1.0/1.1)/Icecast support
	Control and alter sound in different ways
	Processing different streams and formats
	Controls output stream and formats
	Playing mpeg layer 1/2/3
	Playing wav and formats supported by mikmod

Table 4.11: Iteration 1: Application Logic Layer Subsystem Responsibilities

Subsystem	Responsibility
High Level Services	Scroll wheel support
	Saving http streams to hard disk
	HTTP authentication
	Proxy authentication support
	Abstract library, storage, and operating system functionality
Libraries	GUI support
	I/O support
	Abstract low-level OS services
Storage	Abstract low-level OS support for various storage devices
	Provides facilities for storing information like user preferences, playing lists, etc.

Table 4.12: Iteration 1: Service Layer Subsystem Responsibilities

XMMS Subsystem — XMMS subsystem implements basic functionality of the application, and contains basic user interface functionality and management. It has one subsystem called defskin which contains basic user interface skins. The following functionality is supported:

- Displaying user interface widgets in platform independent manner.
- User interface management in platform independent manner.
- System initialization.
- Access to and management of the files at remote locations.
- Interface to different plugins.
- Plugin management.
- General way of handling input and output (specifics are handled by plugins). This is the basic functionality since song playing, stopping, pausing, etc. are the manipulations of the input and output streams.
- Play-list management.
- File system handling.

WMXMMS Subsystem — This subsystem contains platform and window manager specific user interface functionality. It is responsible for all window managers except GNOME.

GNOMEXMMS Subsystem — This subsystem is responsible for GNOME window manager specific user interface functionality.

LIBXMMS Subsystem — This subsystem contains low level functionality for application and user interface management. It is used by all other subsystems, except intl and po, to support their functionality.

INTL Subsystem — This subsystem contains GNU gettext library, which is used for internationalization of the application.

PO Subsystem — This subsystem contains PO files generated using GNU gettext that provide customization of the user interface with different languages (international support).

General Subsystem — General subsystem contains plugins that can be used to control the application in some new way not provided by core functionality. It has three subsystem, and each of them contains functionality of a specific plugin:

- IR Subsystem — This plugin provides functionality needed to control XMMS with TV / VCR / Stereo remote. It has some user interface functionality in order to display “about window”.
- Joystick Subsystem — This plugin provides functionality needed to control XMMS with one or two joysticks. It also has some user interface functionality in order to display “about window”.
- Song_change Subsystem — This plugin provides functionality needed to control XMMS change songs. It also provides user interface for getting info needed to support its functionality.

Effects Subsystem — Effects subsystem contains plugins that are used to alter the sound. It has three subsystem, and each of them contains functionality of a specific plugin:

- Echo_plugin Subsystem — provides echo effect. Contains functionality to display “about window” and to inform user that echo effect is on.
- Stereo_plugin Subsystem — provides stereo effect. Contains functionality to display “about window” and to inform user that stereo effect is on.
- Voice Subsystem — provides way for voice removal from the song. Contains functionality to display “about window”.

Visualization Subsystem — This subsystem contains three subsystems blur_scope, opengl_spectrum and sanalyzer, which contain plugins for special visual effects. All of them are concerned with user interface.

Input Subsystem — This subsystem contains seven subsystems, which contain plugins responsible for processing different types of input streams. Subsystems are cdaudio, idcin, vorbis, mikmod, Mpg123, tonegen, and wav. They contain user interface functionality.

Output Subsystem — This subsystem contains four subsystems, which in turn contain plugins responsible for producing different types of output streams. The subsystems are

disk_writer, esd (eSound Output Plugin 1.0), Solaris, and OSS (OSS Driver 1.0). They contain user interface functionality.

This high-level concrete architecture allows us to make initial estimates of the rationale behind the architecture. It is still too early to make a detailed analysis of the rationale as many aspects of the architecture are still unexplored. The purpose of this initial rationale recovery is to facilitate and improve recovery of conceptual and concrete architecture in the next iteration. As such, during this iteration, we do not spend time on analysis of alternatives, but only on discovery and presentation of major quality attributes and design decisions. We represent this architectural rationale using a table that captures major quality attributes, quality facts (quantified quality attributes), design decisions and their impacts (Table 4.13).

Quality Fact:	Support different window managers
Quality Attribute:	Usability, Portability
Design Decision:	Create different subsystems
Impact:	WMXMMS and GNOMEXMMS Subsystems
Quality Fact:	Isolate low-level common functionality
Quality Attribute:	Modifiability, Portability, Understandability
Design Decision:	Create new library subsystem
Impact:	LIBXMMS Subsystem
Quality Fact:	Separate highly related functionality groups
Quality Attribute:	Modifiability, Extensibility, Work distribution
Design Decision:	Group functionality into different subsystems
Impact:	General, Effects, Visualization, I/O subsystems (high cohesion, low coupling)
Quality Fact:	Accommodate new functionality of existing type
Quality Attribute:	Extensibility, Work distribution
Design Decision:	Plugin architectural style
Impact:	Plugin design of General, Effects, Visualization, I/O subsystems

Table 4.13: Iteration 1: Architecture Rationale

Now that we have an initial understanding of all major aspects of the architecture, we can

perform a more detailed recovery of each one of them using acquired knowledge about others. This is done in the second iteration.

4.2.3 Architecture Recovery: Second Iteration

While the emphasis of the first iteration was more on conceptual architecture and general intuitive understanding of other aspects of the system, the emphasis of the second iteration is on a detailed study of concrete architecture and its rationale.

Conceptual Architecture

The focus of the second iteration of conceptual architecture recovery is on incorporation of feedback obtained through the initial study of concrete architecture. Although only a high-level concrete architecture recovery was performed, there are already significant differences and violations of basic design principles observed:

- User interface and application functionality, in both core parts and plugins, are not completely separated at the concrete subsystem level.
- We did not expect that the application logic layer plugins will have a user interface and therefore we do not have an connector from the Presentation Layer to the Plugins subsystem in the Application Logic.
- Different types of plugin subsystems are not contained in general plugin subsystem, as indicated in the conceptual architecture diagram.

Although some of these decisions conflict with recommended design principle, we do not analyze them at this point, but update our conceptual architecture to reflect concrete as much as possible, while still preserving many of the initial forward design ideas. This will allow at later stages to analyze mistakes in design of concrete architecture and repair them. Current conceptual architecture is summarized in Figure 4.2

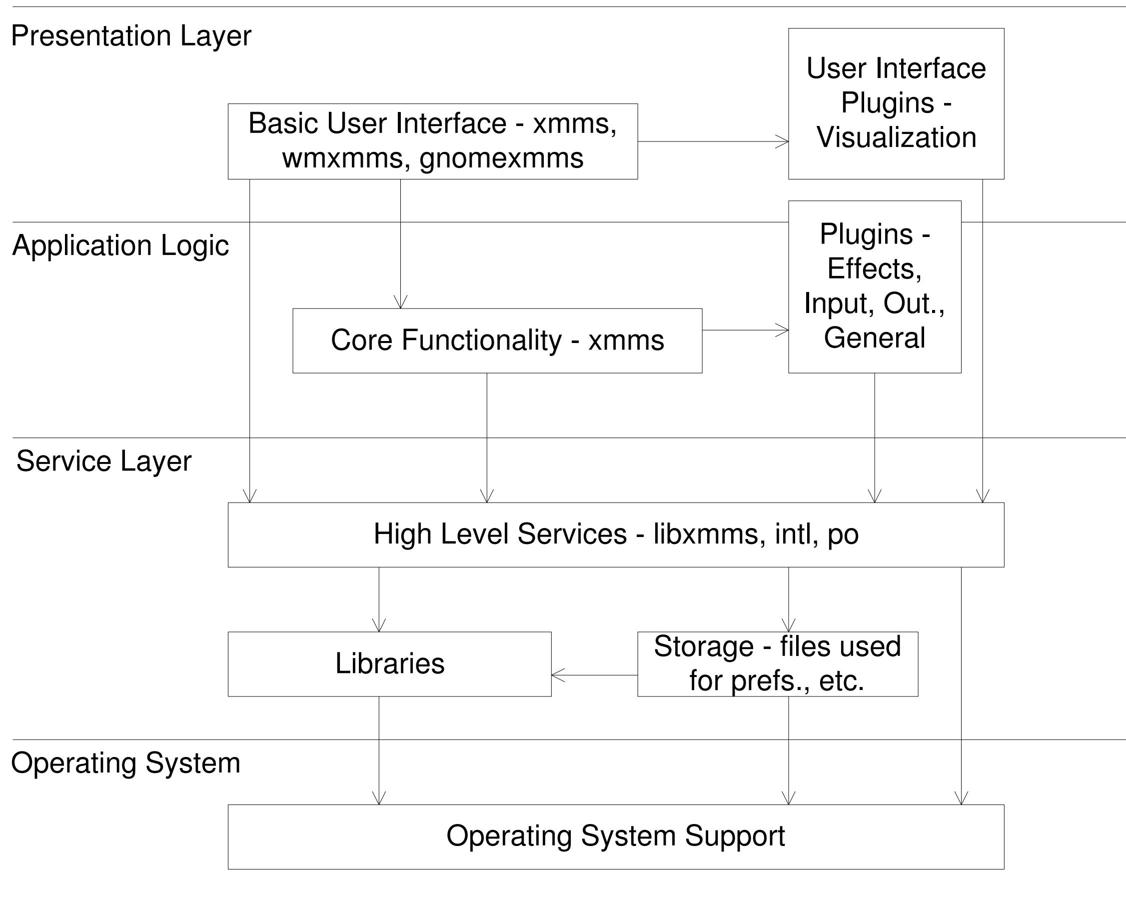

Figure 4.2: Iteration 2: Static Conceptual Architecture

Concrete Architecture

While concrete architecture recovery during the first iteration tended to follow a top-down approach, during the second iteration it followed a bottom-up approach. During the second iteration, analysis focus was on the module level. As XMMS was built using the C language and procedural paradigm, we have chosen to consider a module to be a source code file.

The following activities were performed during architecture recovery at the module level:

1. Architecture presentation and navigation using PBS [PBS].
2. Module analysis using a source code navigation tool Understand for C++ [UND].
3. Responsibility recovery.
4. Use of navigation facilities of PBS and Understand for C++ in conjunction with recovered responsibilities in order to analyze architectural use-cases.

PBS allows us to visualize and analyze the concrete architecture of a system. Figure 4.3 presents the highest level of decomposition of XMMS. Figure 4.4 presents the architecture of the Visualization subsystem and interconnection using two types of connectors. Figure 4.5 presents the Input subsystem with even more connectors. The system can be navigated like this using PBS facilities from one subsystem to another. The lowest level of decomposition presented is at the module level. Figure 4.6 presents the CDAudio subsystem and interconnections among its modules.

Responsibility assignment is performed with the help of a source navigation tool. The goal is to elevate functionality provided by a set of methods, and to present it in a simple and understandable way. The value of this responsibility documentation is in the fact that knowledge about low level facts is preserved when it is of crucial importance for understanding of architecture. It is not needed to document all responsibilities, but the ones that were recovered with architectural goals in mind. For example, if a system is using some functionality from another system, and by original specification it shouldn't, an analyst should perform a low-level analysis of used functionality. Knowledge about that set of functionality may be documented using responsibilities and contracts. Interactions are then presented using architectural use-cases.

Many of the analyzed methods contain functionality that is not architecturally relevant. Nevertheless, in combination with other methods they can provide significant architectural impact,

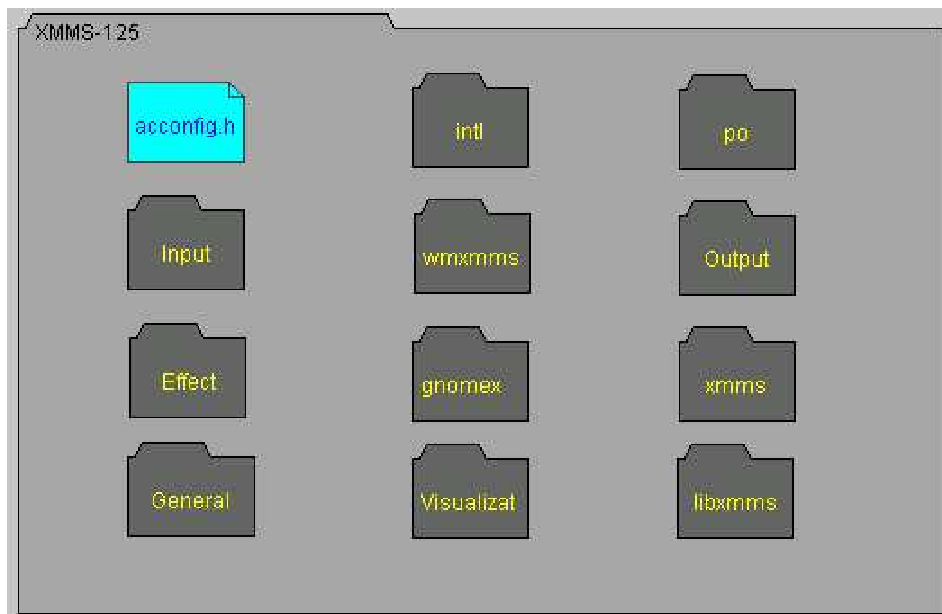

Figure 4.3: PBS XMMS Top Level View

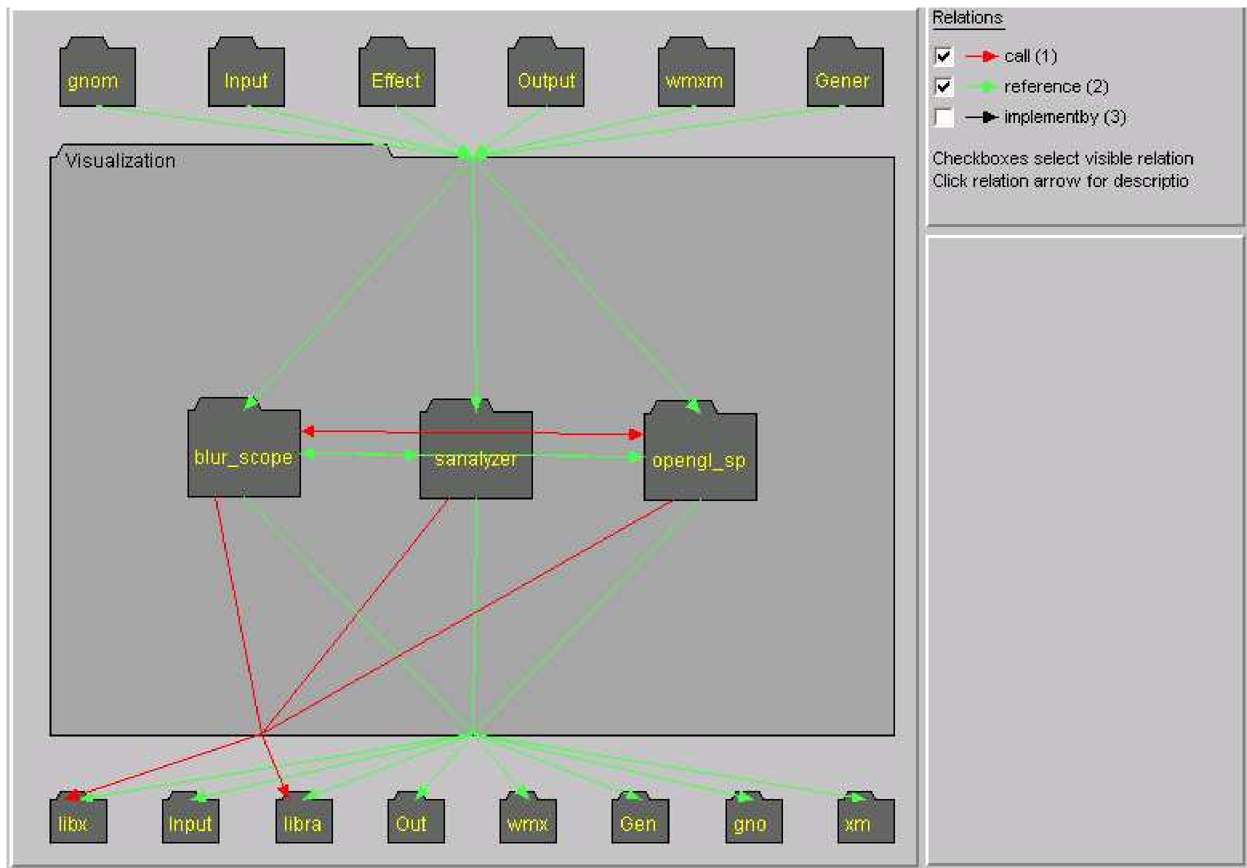

Figure 4.4: Visualization Subsystem

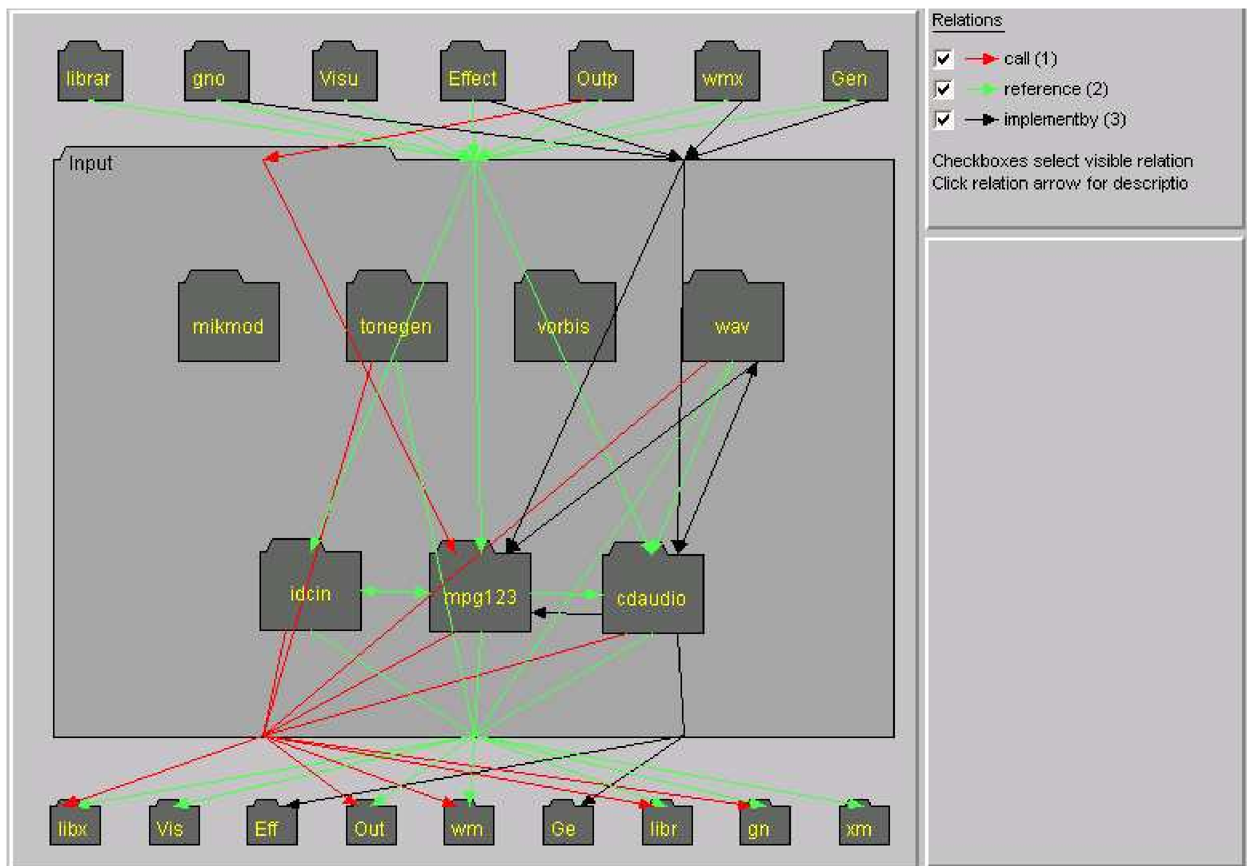

Figure 4.5: Input Subsystem

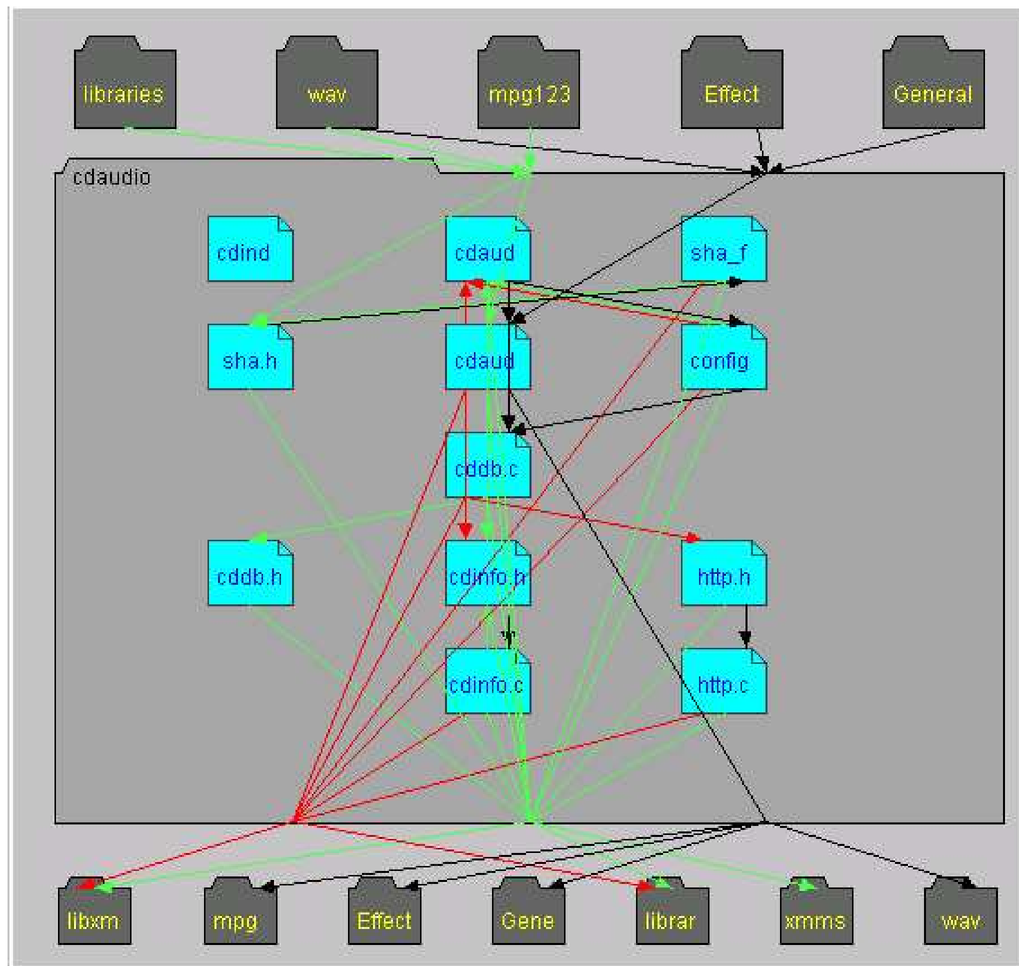

Figure 4.6: CDAudio Subsystem

and if this is the case, then responsibility assignment is performed for a whole group of methods. Table 4.14 presents both types of responsibility assignments.

Responsibility	Methods
Delegate "play" event to all subscribers	void mainwin_play_pushed(void)
Delegate "stop" event to all subscribers	void mainwin_stop_pushed(void)
Delegate "eject" event to all subscribers	void mainwin_eject_pushed(void)
Adjust sound effects	void mainwin_adjust_volume_motion(gint v) void mainwin_adjust_volume_release(void) void mainwin_adjust_balance_motion(gint b) void mainwin_adjust_balance_release(void) void mainwin_set_volume_slider(gint percent) void mainwin_set_balance_slider(gint percent)
Get configuration info	void read_config(void)
Set configuration info	void save_config(void)
Initiate GUI setup	void setup_main_window(void) void draw_main_window(gboolean force) void draw_mainwin_titlebar(int focus)

Table 4.14: Method-Level Responsibilities (Main Module)

It is also useful to elevate responsibilities to the module level (*i.e.,* to clarify responsibilities of a module as an architectural entity). These module-level responsibilities are sometimes equivalent to a set of method level responsibilities, and sometimes are an abstraction of them. The choice depends on the level of analyst's desire to hide internal details. Table 4.15 presents a set of module level responsibilities.

Similarly, responsibilities can be abstracted to the subsystem level. While module-level responsibilities provide a compact way to encapsulate and represent architecturally significant features, method-level responsibilities are used to understand and present module and subsystem interactions using architectural use-cases. The advantage of architectural use-cases over other presentations like sequence diagrams is that they present dynamic aspects in a comprehensible way while hiding low-level details.

Responsibility	Module
Manage user and system configuration	libxmms/ConfigFile.[c,h]
Manage system directory access	libxmms/DirBrowser.[c, h]
Abstract low-level OS functionality	libxmms/util.[c, h]
Provides basic XMMS functionality	libxmms/xmmsctrl.[c, h]
Setup control session	xmms/controlsocket.[c, h]
Setup dock windows	xmms/dock.[c, h]
Equalizer user interface	xmms/eq_graph.[c, h] xmms/eq_slider.[c, h] xmms/equalizer.[c, h]
"Hint" user interface	xmms/hints.[c, h]
Play-list popup window	xmms/playlist.[c, h] xmms/playlist_slider.[c,h] xmms/playlist_popup.[c, h] xmms/plsylistwin.[c, h]
Decode a file at url into a proper format	xmms/urldecode.[c, h]
Abstract low-level OS functionality	xmms/util.[c, h]
Integrate skins into XMMS user interface	xmms/skin.[c, h] xmms/skinwin.[c, h]

Table 4.15: Module-Level Responsibilities

Architectural use-cases are built using navigational capabilities of tools like PBS or Understand for C++ [PBS, UND] in conjunction with documented responsibilities. The main value of this approach is not in a documentation of all possible use-cases, but in an ability to recover them as needed. Although responsibilities do not have to be documented within source code, the advantage of having them documented there is that a transition from architectural level analysis to low-level design analysis is seamless. Bellow is an example of a fully developed documentation architectural use-case:

- *Name:* Play Song
- *Actors:* End-user
- *Stakeholders:*
    1. End-user
    2. Music provider
- *Event:* User pushes play button
- *System:* xmms, libxmms, input, output, visualization
- *Purpose:* Describe collaboration among subsystems to accomplish “play” functionality
- *Priority:* 10/10-core business process, crucial for business operation
- *Overview:* Input stream is processed to produce wanted output (song playing or streaming to a file on hard disk)
- *References:* None
- *Related Use-Cases:* Setup
- *Responsibility:* Play media stream or write it to a file
- *Preconditions:* Play-list was configured, Setup use-case successfully performed
- *Postconditions:* System stops playing, after input stream end, if Repeat option is turned off.

- *Invariants:* None
- *Main Scenario:*
  1. xmms: User interface component signals "play" event is raised.
  2. xmms: Signal to input subsystem to start processing data
  3. xmms: Connector between input and output is established
  4. input, output, visualization: Start processing data streams
  5. input: If end of the stream signal xmms and stop
  6. xmms: If "Repeat" option turned on signal input to start processing again, else "stop" signal to output and visualization
- *Alternatives:*
  - step 4: data stream disconnected before end of it (file deleted, network connection went down, etc.) — raise exception and inform user
  - step 4: special effects events raised — activate appropriate plugin, which alters output
- *Quality Attributes:*
  1. Responsiveness — events are handled without delays
  2. Reliability — user is informed within 1 second if system stops due to data stream problems
- *Technology:* Network access support for network streams
- *Special Requirements:* For low-end systems, output processes have higher priority over visualization and affect subsystem's processes
- *Open Issues:* None

We used single column format for our use-case. One could also use multiple column format to emphasize subsystems and modules. The second option has a drawback that it is harder to format text properly thus increasing production and maintenance time.

Architecture Rationale

Detailed concrete architecture analysis during the second iteration gave us additional insights into architecture rationale. Additional quality facts that were discovered are of a negative nature (*i.e.,* ones that contradict generally accepted design practices). Table 4.16 presents rationale for architectural design of our system. The first two items are new ones, discovered during the second iteration of concrete architecture recovery.

The first two potentially negative design decisions contain a "Note" section indicating that architecture repair is recommended. Also, they contain an "Alternatives" section that indicates possible alternative decisions and repair approaches. These repairs can be performed during special refactoring sessions, or as a part of major new feature introduction.

This concludes our second iteration, and main architecture recovery efforts. Additional iteration could be used in order to refine different recovery artifacts, or to achieve some other purpose, such as preparation of produced artifacts for a particular audience.

Our recovery process had two main iterations, but for a larger system there can be more iterations depending on the number of people working on system recovery, quality of tools used, etc. While activities presented in the first iteration can be performed by a small dedicated team of analysts, activities from the second iteration are best done by actual developers of the system. Responsibility instrumentation can be performed during actual development with very small overhead. This not only serves for architecture recovery, but also improves general code understandability and quality. These results can then be used by analysts to efficiently recover interesting architecture facts.

4.3 XMMS: Focused Evolution Recovery

In order to even further practice and validate our process, we applied it to the recovery of several releases of the XMMS system in order to understand and analyze the changes in the architecture over time. Following presents the obtained results, and emphasizes the evolution of the static structure of the system and its correlation with the other architecture evolution aspects.

Product Name and Characteristics XMMS Player — Unix application influenced by its Windows counterpart WinAmp.

Quality Fact:	No clear separation of basic GUI functionality from application logic
Quality Attribute:	Efficiency (at the expense of Modifiability and Understandability)
Design Decision:	GUI and application logic bound tightly together
Impact:	xmms subsystem contains both GUI and application logic
Note:	Architectural repair recommended
Alternatives:	Isolate GUI functionality into a separate subsystem
Quality Fact:	Plugins contain GUI functionality
Quality Attribute:	Work distribution (Limits portability and modifiability)
Design Decision:	GUI functionality embedded within plugins
Impact:	Plugin subsystems contain both GUI and logic
Note:	Architectural repair recommended
Alternatives:	Delegate GUI functionality to main GUI subsystem (xmms)
Quality Fact:	Support different window managers
Quality Attribute:	Usability, Portability
Design Decision:	Create different subsystems
Impact:	WMXMMS and GNOMEXMMS Subsystems
Quality Fact:	Isolate low-level common functionality
Quality Attribute:	Modifiability, Portability, Understandability
Design Decision:	Create new library subsystem
Impact:	LIBXMMS Subsystem
Quality Fact:	Separate highly related functionality groups
Quality Attribute:	Modifiability, Extensibility, Work distribution
Design Decision:	Group functionality into different subsystems
Impact:	General, Effects, Visualization, I/O subsystems (high cohesion, low coupling)
Quality Fact:	Accommodate new functionality of existing type
Quality Attribute:	Extensibility, Work distribution
Design Decision:	Plugin architectural style
Impact:	Plugin design of General, Effects, Visualization, I/O subsystems

Table 4.16: Iteration 2: Architecture Rationale

Product Family and Characteristics Multimedia players family. Main characteristics are:

- Stand-alone desktop applications.
- Layered architecture.
- Extensive GUI and graphical support and capabilities.
- Image and audio stream processing.

Development Process Main characteristics of XMMS development process are:

- Small group of developers.
- Open-source development model — Extensive testing community.
- Development driven by requirements of other systems (WinAmp in particular).

Target Market Main market characteristics are:

- Software provided for free.
- Community provides extensive feedback for system development.
- Strategic goal is to strengthen position of Unix based systems in desktop market.

Stakeholders and Attributes *See Section 4.2.2.*

Architecture *See previous sections of this chapter.*

Measurement Method We have analyzed 12 releases of XMMS, which covers a period of four years of development.

Business Attributes Business attributes have stayed stable over time. For discussion of business attributes, see Section 4.2.2.

Project Management Attributes Project management attributes have stayed stable over time. For discussion of project management attributes see Section 4.2.2.

Architecture Attributes We analyzed incorporation, removal, and repair of modules and subsystems over 12 releases of the system. For every architecture modification, we describe the change and the rationale behind it:

Release 0.9.0 to 0.9.1 No architectural changes.

Release 0.9.1 to 0.9.5 Architectural changes:

- New Visualization module — Provides interface and uniform way for management of all visualization plugins. This improves customizability and modifiability of the core of the system, and ease of plugin development. Visualization module has a role of connector in the system.
- New fft module — Provides uniform way for frequency analysis and is used by Visualization module to support oscilloscope functionality.
- New high level Visualization subsystem — Isolation of different visualization plugins from the rest of the system. This improves modularity, understandability, and modifiability of the system.
- Three new visualization plugin subsystems and modules (blur_scope, sanalyzer, sanalyzer)
- New idcin input subsystem and plugin — Provides a major new functionality as it incorporates video playback into what was only audio player up to that point. This had impact on business attributes and was a source of possible architecture drifts in future.
- New mpg123 subsystem and plugin — Provides support for mpg files.

Release 0.9.5 to 0.9.5.1 Minor architectural change — addition of a module to blur_scope plugin.

Release 0.9.5.1 to 1.0.0 Architectural changes:

- New fullscreen module — Provides full-screen support for visualization plugins (libxmms subsystem).
- New dga module — Provides DGA support for full-screen mode (libxmms subsystem).
- New cdaudio plugin — Provides CD audio streams support.
- New dock module — Provides "docking" ability to GUI windows.

Release 1.0.0 to 1.0.1 No architectural changes.

Release 1.0.1 to 1.2.0 Architectural changes:

- Architectural repair — Move of fullscreen and dga modules from libxmms to xmms subsystem. Rationale is that both these modules contain high-level functionality, which is more appropriate for xmms subsystem than for low-level library subsystem, where it was assigned at first.
- Architectural repair — surround_echo subsystem merged with Echo plugin. Rationale is that there exist high-coupling between features from these two modules.
- New urldecode module.
- New internationalization support subsystem intl and i18n module.

Release 1.2.0 to 1.2.1 No architectural changes.

Release 1.2.1 to 1.2.2 Minor architectural change — addition of formatter module to libxmms subsystem.

Release 1.2.2 to 1.2.3 No architectural changes.

Release 1.2.3 to 1.2.4 Minor architectural change — addition of vorbis plugin to input subsystem.

Release 1.2.4 to 1.2.5 Architectural changes:

- New solaris output plugin — Increases portability of the system to Solaris platform.
- Input vorbis plugin is greatly enhanced — several new modules added.
- DGA and i18n modules are removed — removal of obsolete features.

Conflicts None.

Selection Factors the WinAmp system has established a standard in the area of audio-multimedia applications. Most users expect to see the same or superior set of features in similar multimedia systems. XMMS has successfully provided this set of features, compared to other Unix-based multimedia systems, which made it the leading multimedia player for Unix platforms.

The main conclusion of our architecture evolution analysis is that XMMS vision and architecture have successfully accommodated many new features over time even though these were large,

architecturally significant changes. This is due to its highly modularized plugin architecture, and business vision whose goals have been already proven on another system (WinAmp). The main drawback of the initial architecture — thick GUI layer did not impose evolutionary problems due to the fact that most changeable application logic and GUI functionality were isolated into their respective plugins.

4.4 Summary

In this chapter, we have presented one possible instance of the application of our process. We conducted a successful architecture recovery for a system of a nontrivial size. This recovery has improved our confidence in the effectiveness and the applicability of our process. Building on the knowledge presented in the previous chapter, we elaborated on the particular steps by providing sample low-level recovery activities that can be used for the recovery of architectures of other systems.

In the next chapter, we present the evaluation of our approach in the context of our case study.

Chapter 5

Evaluation

In this chapter, we present an evaluation of our process and the results obtained through its application to the recovery of the XMMS multimedia system's architecture. Because the observations about the characteristics of the process depend directly on the type of the chosen candidate system and the conditions under which the process is applied, we first discuss the case study issues and validity. Also, since our process is based on an already existing architecture recovery approach, we discuss the properties of this recovery approach. We then evaluate the project management issues that control the overall execution of our process, allowing an evaluation of the particular process activities. We also evaluate the particularities of the main techniques used during the recovery process. The main emphasis is on the observed problems and the possible solutions to them.

5.1 Case Study Validity

In order to improve the validity of the results derived from the conducted case study, we have taken several concerns and issues into consideration when choosing our candidate system. These concerns can be divided into two categories. The first one is concerned with inherent properties of the chosen system as they relate to our process (*e.g.,* the size of the system and the number of development years). We will refer to these as system concerns. The other group is concerned with issues that relate to the analysts' relation to the system under consideration. This group includes issues such as analysts' familiarity with the system's problem domain, experience with

programming languages used in system's development, and so on. We will refer to these as analyst concerns.

5.1.1 System Concerns

The main issues from the system's perspective, when choosing our case study example, were:

- The size of the system was approximately 65,000 lines of code, which places it into a mid-sized systems category. This was of importance to the validation of our process as the process's main use is for recovery of small to mid-sized applications. Also, many large systems usually have well defined subsystems of size similar to that of our system. These subsystems can be studied relatively independently, thus permitting the use of our process in such cases too.
- The XMMS system was under development and effective use for approximately 5 years. This was important for validation of our process since we were dealing with a relatively stable architecture, which has evolved over time. This implied that we had a possibility to detect features such as obsolete functionality, changes in business goals, and so on.
- The core development team consisted of 3 developers, with contribution from many third parties, and a large feedback from the user community, which is typical among widely used open-source systems. The size of the team was important to insure that the architectural influences did not come only from one source. This conforms to the reality of larger scale development and affects the quality and stability of the architecture.
- There was no architectural documentation. In order to increase the difficulty of the case to which our process was applied to, we chose a system for which there was no architectural documentation. This was done with the assumption that it is harder to perform the architecture recovery of a system for which no architectural information is provided.
- There were many contributions from third parties. We wanted a system that was not developed in isolation, as many of the architectural issues arise when it comes to interoperability, distributed development, and similar. There were many add-ons and pluggins, developed by third parties, which provided many sources of architectural influences and made a more realistic case for the architecture recovery.

5.1.2 Analyst Concerns

Since many of the aspects of the process directly depend on the analysts' capabilities and familiarity with system, we had to take these into consideration when choosing the system. In order to expose as many process's drawbacks as possible, we had to find a candidate system which is challenging enough from the analyst's perspective. The main concerns and issues that we took into account were:

- Problem domain — The analyst that performed architecture recovery, the author of this thesis, was familiar with the problem domain only from the user perspective. This familiarity was only from the perspective of using WinAmp, the system which XMMS was modelled after, for solely playing music streams. He was not familiar with any other features, such as writing streams into files, nor with many external issues like different types of multimedia files, different compression algorithms, and so on. This unfamiliarity with the problem domain increased the difficulty of architecture recovery, and put the process on the test under the very difficult conditions, which can still occur in reality (*e.g.,* when a new member joins the development team, and has to quickly get up to speed without the input from other team members).

- Programming languages — As many of the aspects of the recovery depend on the analyst's ability to understand the underlying code organization and code itself, it was important to choose the system which was developed using the languages that the analyst was relatively unfamiliar with. XMMS was developed using C language. The analyst had a theoretical background in C through the study and use of C++ language, but has never developed any larger system using C language. This again falls into a relatively difficult case, from the process perspective, because in most situations a better familiarity with the programming languages is needed in order to join a development team.

- Architecture design and analysis — The analyst had a solid theoretical background in architectural analysis and design, and was familiar with and had access to many techniques and patterns used for analysis and design of software architectures. Nevertheless, the analyst did not have a practical experience in the design of a system of size and type similar to the one of the size of XMMS system. This was again a less than ideal situation for the application of the process, which allowed us to test it under such conditions.

- Previous experience in architecture recovery — Prior to the architecture recovery of the XMMS system, the analyst had experience with recovery of architecture of a SCSI driver subsystem of the Linux operating system kernel. This was legacy code of the size of approximately 300,000 lines of code, and was recovered using a different approach than the one presented in this thesis. This allowed us to test our process from a perspective of an analyst relatively familiar with architecture recovery issues and problems, from both theoretical and practical viewpoints, and to compare our approach to another.

As we have seen, we have chosen the case study such that we were able to validate our approach under difficult conditions, which are still accountable in production environments. This was done under the assumption that if the process works under these worse than typical conditions, it will be even more usable under typical production environment conditions, and allow us to spot as many problems and drawbacks as possible.

5.2 PBS: Reference Process

The first recovery of the XMMS architecture was performed using the PBS approach [PBS]. The PBS approach is the set of activities performed in order to arrive to the set of artifacts imposed by the PBS tool. This process is similar to the processes used with similar tools such as Rigi and Shrimp [Rig, Shr]. The PBS process can be summarized in three main steps:

1. Derive and present the conceptual architecture.
2. Derive and present the concrete architecture of the system.
3. Explore the differences between the conceptual and the concrete architecture.

Our process is inspired by and builds on the PBS approach. In this section we present basic, high-level issues that we wanted to improve on, in relation to the original PBS approach, and difficulties that we had during the first system recovery and artifact analysis.

The main problems that we observed during the application of PBS approach were:

1. The PBS approach did not specify a repeatable set of steps that have to be performed in order to arrive at conceptual and concrete architecture, so that they can be routinely performed in a production environment.

2. The PBS approach did not specify what constitutes a set of artifacts that form the conceptual architecture.

3. The PBS approach did not make a clear distinction between different architectural aspects (*i.e.,* the conceptual architecture relates to the logical view, while the concrete architecture focuses on the development view of the system).

4. The PBS approach did not include any artifacts that presented architecture rationale.

5. The PBS approach did not recommend the preservation of the problem domain knowledge, which is unavoidably generated during the architecture recovery.

Since the PBS approach does not specify either the steps to be performed nor the exact set of artifacts to be produced (for conceptual architecture), there is a wide variation in the quality of the analysis, depending on who performs it. This is not to say that the clear definition of these steps would completely eliminate these difference, but it would definitely reduce them. Also, it would be less difficult for other interested developers to analyze and relate different produced artifacts.

The clear distinction between the views represented by the conceptual and concrete architecture is needed in order to reduce ambiguity, improve understandability, and describe the actual separation of concerns. As the conceptual architecture is derived from high level sources such as documentation, feature lists, stakeholder interviews, etc., it tends towards encapsulating logical aspects. On the other hand, the concrete architecture, is derived mostly from the source code, and it tends towards the presentation of the development view aspects, such as source code organization, source code dependencies, etc. Finally, both conceptual and concrete architecture can be viewed as the ways of representing a unified logical and development view at different levels of abstraction.

The recovery of rationale and problem domain aspects are unavoidable activities during an architecture recovery. Since the PBS approach specifies neither techniques nor artifacts that would preserve the facts about rationale and problem domain, the information about them is obtained and lost during the actual process. The artifacts about the problem domain and rationale would improve the understanding and usefulness of generated architecture recovery artifacts. The problem domain represents the context from which architecture is derived, and rationale presents why the architecture is as it is (*i.e.,* it is the source of the architectural design decisions).

5.3 Project Management Issues

In this section, we evaluate our process from the project management perspective. The project management issues affect the overall organization of the process and the execution of the particular process steps. As such, it is important to evaluate them, especially with the reference to the original PBS approach, since the issues such as the number of recovery iterations contribute to the quality and completeness of produced artifacts.

5.3.1 Process Iterations

The first problem that we had during the initial recovery of the system was the large amount of information to be processed. We tried to recover the conceptual architecture completely before embarking into the recovery of the concrete architecture. Also, we tried to recover the architecture at different levels of abstraction without making a clear distinction among these levels. This led us to the following problems:

- We had hard time organizing and presenting this large amount of information.
- We were not able to clearly separate different aspects of the system, and to know what information to reject and what to retain.
- We were unable to track the reasons for certain architectural decisions, as the aspects at different abstraction levels collided thus producing the conflicting underlying reasons.

In order to deal with this problem, we tried to perform the iterative recovery of the conceptual and the concrete architecture separately. This posed a problem since the interconnections between the conceptual and the concrete architecture are high. Therefore, we finally took the approach of iterative recovery, as presented in the case study, which was based on the iterative recovery of both conceptual and concrete architectures in parallel, at the different levels of abstraction. In addition, we used the aspect separation in order to improve the clarity of presented results and to improve the potential for the rationale recovery.

The issue that is still unresolved is the existence of a clear set of heuristics that would help us evaluate when we should perform a transition from one iteration to another and how many iterations should be performed all together in order to improve efficiency and minimize costs. In

order to establish such a set of heuristics there have to be more case studies performed, and all the recovery techniques stabilized and automated as much as possible. Also, these would depend on the type of underlying development technology and the type of application problem domain.

5.3.2 Individual and Team Issues

Another unresolved issue that is common to both, our process and the original PBS approach, is the team collaboration. None of these approaches specifies the protocols, additional steps, and transitional artifacts required in order to efficiently and routinely integrate the recovery artifacts. This is of lesser importance in the case of our process as it is intended for the recovery of the architecture of small to medium-sized systems, while the PBS approach also targets large systems. According to our experience and estimates, the architecture of systems of the size similar to the system from our case study can be recovered by small teams, up to three people, in four to twelve weeks. For such small teams to communicate, the introduction of formal communication protocols and artifacts is not necessary, and direct informal communication can be considered as the best choice. On the other hand, if our process is to be adapted and used for the team recovery of large systems, there must be a more systematic way to manage the work of the team members and unify the produced artifacts.

The important issue, at the individual level and thus indirectly at the team level, comes from the fact that the process depends on each member's different architectural skills and abilities. The required background for the architectural recovery is the knowledge about architectural and design patterns, the ability to perform forward architectural design, knowledge of system's underlying technologies, and so on. Due to this dependency of the process on the individual skills, the quality and the focus of the recovery artifacts varies.

The author's background, as related to XMMS related technologies and architecture in general, at the time the case study was performed was:

- Beginner level of familiarity with C language and Unix programming.
- Good knowledge of architectural and design processes and techniques, with lack of the practical experience in the design of the mid-sized and large software systems.
- Experience in the architecture recovery of one large system, and familiarity with experi-

ences of other people who used PBS approach for the large systems' architecture recoveries.

This gives us confidence that our process can be used by developer's with intermediate skills, as well as with advanced. Also, the wide familiarity with techniques used for recovery within software engineering community and the relative ease of learning makes it possible for beginners to become productive within a short period of time.

5.4 Process Step Issues

The focus of this section is on describing our experience and problems during the execution of each process' activity. We do not discuss in this section the issues of particular techniques used to achieve particular process goals and the issues related to the produced artifacts. These are discussed in the following two sections.

5.4.1 Project Elaboration

As we have discussed at the beginning of our case study, we did not perform a detailed analysis of the feasibility of our project and the effort that will be required. Like with all software project management estimations, the precise estimation of the cost of the reverse engineering project is difficult to achieve. Since our process was primarily designed to be used for the recovery of small to mid-sized systems, this estimation is not of the crucial importance — our estimate was that the upper time limit for the recovery of the architecture, for a system of the size similar to the size of XMMS, is one month for a team of three analysts. On the other hand, if our process is to be adapted and used for the recovery of architecture of large systems, one should incorporate cost estimation techniques. The original PBS approach had the same problem because it lacked any project elaboration phase.

We presented some of the influences on the cost of the recovery at the beginning of our case study. Out of the ones that we mentioned (documentation, quality of code, size of the system, developer's information, and analyst's problem domain knowledge), we found that the analyst's previous knowledge about problem domain and related applications plays a dominant role over the others. Even in the case that all the other influences are positive, it is very hard to recover

the architecture if one cannot put the gathered information in the context of the problem domain; and it is very time consuming to learn details about the problem domain.

5.4.2 Analysis of Existing Artifacts

In order to simplify the analysis of the existing documentation, we performed an extraction and classification of the potentially relevant architectural concepts and features. In order to perform this extraction, we needed background knowledge about the properties of functional and non-functional requirements as they relate to the software architecture. Our main focus was on the analysis of different requirements documents.

Although we had access to and analyzed only a list that only summarized the high-level features of the system and user manual, for larger systems one usually has access to other kinds of requirements documents. These include software requirements specification, requirements rationale, data dictionary, etc.

5.4.3 Domain Architecture

When we attempted to extract the architecture of XMMS using the original PBS approach, we did not have any intention to recover the architecture of the problem domain. The problem that we had was that it was not possible to extract the conceptual and concrete architecture without having to search for clues in the problem domain. This resulted in an unorganized exploration of the problem domain concepts, and this exploration was not as efficient as it could be as we were not sure what to look for. Also, we did not make any attempt to preserve this gained problem domain knowledge, which could be reused in many different ways in the future development of the system. Therefore, one of the goals of our new approach was to deal with domain architecture issues.

The first approach to solve this problem focused on only documenting the facts about the problem domain. This did not prove to be an effective approach, as the collected facts were simply listed, without a particular organization, and as such they were not very useful for the recovery of conceptual and concrete architectures. It was difficult to relate one fact to another, and to further relate them to the architectural artifacts. Also, we were not able to distinguish different types of concepts that occur in the problem domain.

In order to solve these problems, we had to introduce a meta-model for our domain architecture, to find a format which will present these fact so that they are directly usable for the recovery of the conceptual and concrete architectures, and to find a systematic way to analyze the domain in the early stages of our process. We solved this problem as described in Chapter 3, and using the proposed steps, successfully recovered and presented the domain architecture of XMMS. We used these recovered domain concepts also to guide the recovery of the other architectural aspects.

One of the drawbacks is that we did not make additional attempts to produce alternative meta-models and formats. This is not to say that our approach is the only one or the ideal one, but it was successful in recovering and organizing these artifacts compared to our initial attempts. There are other possible ways to approach this problem. For example, if we are working on the recovery of the architecture of a system for which there are already existing business domain analysis documents, possibly the most efficient way is to follow the already existing meta-model and artifacts.

5.4.4 Conceptual Architecture

The first attempt for the recovery of the conceptual architecture using the original PBS approach was successful only as far as the static conceptual architecture was concerned. Although not clearly required, we attempted the recovery from the logical perspective even at the early stages of our recovery. This resulted in a static conceptual architecture that did not change drastically during the first iteration of our new approach.

The first disadvantage of the PBS approach is that it did not require any form of the dynamic conceptual architecture recovery. This led to the discovery and presentation of structural components without any emphasis on how they cooperate in order to achieve the overall goals. This lack of analysis of the dynamic aspects of the architecture resulted in a reduced understanding of the underlying reasons of the particular static system decomposition. Our approach tackled this problem of the dynamic architecture using the architectural use cases.

The advantage in using the architectural use cases for the presentation of dynamic properties is in the fact that the basic building blocks of these use cases, subsystem responsibilities, are directly derived and produced during the work on the domain architecture artifacts and during the static conceptual architecture recovery. Therefore, the early iterations of our process focused

on the subsystem responsibility assignments. On the other hand, later stages focused on the integration of these responsibilities into fully developed architectural use cases.

We found that the particular strengths of the architectural use cases are as an abstract model of the communications that occur within the system and a technique that drives the recovery of the architecture and integrates different architectural aspects. Nevertheless, we found that it is too tedious to document all the use cases manually. Also, we found appealing the use of use case formats other than the one presented in the case study.

Problems that we had during the initial architecture recovery were the integration of conceptual and concrete architectures and the refinement of conceptual architecture based on the results obtained during the recovery of the concrete architecture. Since the original PBS approach was not iterative, the analyst was aiming to recover as good conceptual architecture as possible, followed by the recovery of the concrete architecture. This often resulted in the architectural presentations that are not as focused as they could be and with a large distance between the concrete architecture and the conceptual architecture. In order to deal with these problems, our process encourages the iterative refinement of both kinds of architectures, thus allowing the input of the facts discovered during the concrete architecture recovery into the conceptual architecture. This is in conflict with the PBS approach in which the conceptual architecture is derived only from sources other than source code.

We have found that this iterative approach simplifies the recovery process, improves the understanding of particular architectural aspects of the system through the indirect analysis of the dependencies among them, and improves the quality of the presentation. Also, the attempt for bridging the distance among different aspects and unification of views simplifies the understanding of the system, and allow us to present its architecture from different perspectives, *e.g.,* pure conceptual logical perspective, pure concrete development perspective, unified logical and development perspectives, etc.

5.4.5 Concrete Architecture

One of the main strengths of the PBS approach is the recovery of the static concrete architecture. As this step of the overall approach is almost completely automated by the PBS tool, our new approach adopted the PBS approach static concrete architecture recovery techniques without significant changes. The main enhancements were introduced because of the need to incorporate

and improve systematic dynamic concrete architecture recovery.

As with dynamic conceptual architecture recovery, the PBS approach does not emphasize dynamic concrete architecture recovery. In order to overcome this problem, we introduced the activity of the responsibility assignment to the modules during the recovery process.

The first attempt that we made in order to recover the concrete architecture was focused on the immediate recovery of the facts in a bottom-up fashion. We approached the analysis of the concrete architecture starting directly from the analysis of the source code. This quickly proved to be an inefficient approach because we had difficulty in relating the discovered concepts due to the large amount of recovered information. In order to deal with this problem, we decided to try to approach the problem first in a top-down fashion and then in a bottom-up fashion. This led to the analysis of the source code structure down to the level of modules, and an attempt to relate them to the previously recovered conceptual architecture artifacts. Following that, we used this high-level concrete architecture in order to organize and focus on the recovery of the low-level concrete architecture. This recovery included the analysis of the structure at the function level and the recovery of concrete responsibilities, which were related to the actual functions and modules.

This approach resulted in a static concrete architecture enhanced over the concrete architecture recovered using the PBS approach. This enhancement was in the detailed specification of responsibilities of each module and their relation to the functions contained within each module. Also, during the process, we elevated the abstraction level of the module responsibilities from function level to the module level. This resulted in an improved analysis of the concrete dynamic architecture of the system.

After the extraction of the architectural facts using the PBS and Understand for C++ systems, we used them to navigate the source code, and together with previously documented responsibility information, generate the architectural use cases. We discuss the details of use-case recovery and PBS issues in section 5.5.

5.4.6 Architecture Rationale

As the primary goal of our process, successful architecture rationale recovery depends directly on the successful recovery of other artifacts. As such, we attempted to recover and present all other architectural artifacts in a form that simplifies and facilitates rationale recovery as much as

possible.

Architecture rationale recovery is the most subjective recovery activity performed as a part of our process. While other techniques are mostly of an observational nature, rationale recovery relies upon a large amount of background architectural knowledge. Also, it is one of the components of the process that is very hard to automate in any way. This leads to the situation that successful recovery of rationale depends largely on the capabilities and performance of the individual analysts.

This subjectivity of architecture rationale recovery makes it hard to evaluate. This includes the choice of artifacts, comparison of the quality of produced artifacts obtained using different techniques, performance of different recoveries by different analysts, and so on. Nevertheless, we describe the issues that arose during our recovery of the architecture rationale of XMMS system.

Our rationale recovery approach relies on the iterative analysis of the design decisions that are extracted from the recovered architectural artifacts and the motivations that usually lead to these design decisions. As a major group of these motivations, we have the quality attributes and related theory. We focused on these quality attributes as they are recognized as the major source of architectural decisions.

During the early stages of the rationale recovery, we had a problem with detecting these architectural attributes. This was due to the fact that we were trying to discover the quality attributes directly from the observed design decisions. As the gap between them is large, *i.e.,* a set of quality attributes can be represented using many architecture design decisions, the mapping is often not clear. In order to deal with this problem, we introduced the quality facts, as an intermediate instantiation and a representation of the quality attributes. Although these quality facts simplified the process and improved the efficiency of mapping from architectural decisions to quality attributes, they did not completely remove the ambiguity in some cases. Despite this remaining ambiguity, we were successful in recovery of architecture rationale using this technique, and therefore we did not invest further efforts to find other solutions which would simplify the recovery even more.

5.5 Process Technique Issues

While our process does not prescribe a definite set of techniques to be used for the recovery of the different aspects of the software architecture, we would like to emphasize the issues related to the concrete architectural use-case recovery and the concrete static architecture extraction using the PBS system. These issues especially concern the automation of these tasks.

Common to both techniques, the concrete use case and the concrete static architecture recoveries, is the aim to make them as automated as possible. This automation is very hard to achieve. While our use-case recovery was performed without the tool support, the static architecture recovery was performed using the PBS system. As the crucial for the success of both of these recoveries is the ability to extract interdependencies among the modules, we will focus on the problems specific to the PBS approach and relate them to the use-case recovery.

The two activities that are performed in order to extract the static architecture of the system using PBS are:

1. specification of the modules and subsystems, and

2. extraction of the dependencies among these modules.

These two activities are also needed for the automated recovery of the architectural use cases.

The main problem is in the specification of the modules and subsystems. Currently, it is not possible to extract this information automatically. We need to specify the modules and subsystems manually, and to feed this information into the tool. All other processing, such as the module interdependency recovery, depends on this specification of the modules. As this manual specification of the modules largely depends on the conceptual view of the architecture, it follows that the concrete architecture in reality depends on the recovered conceptual architecture, and as such contradicts the essential notion of concrete.

Although this dependency of the concrete aspects of the architecture on the conceptual ones is evident, the effort invested in the recovery and the refinement of the concrete architecture pays off in the long term. This payoff is the detection of the actual dependencies among the modules and the presentational aspects of the concrete architecture.

In the next chapter, we summarize the benefits of our process, with a reference to the original reference approach, and discuss the problems and obstacles that we encountered.

Chapter 6

Conclusions and Future Research

In order to understand and integrate legacy systems into new environments, and to successfully develop applications for rapidly changing business domains using agile development processes, we developed and presented a lightweight architecture and evolution recovery process. Our approach was based on attribute theory, as that theory permitted us to systematically tackle and recover the rationale behind architecture and evolution.

In Chapter 3 we presented a step by step process for the architecture recovery. We introduced all the techniques that our process relied upon, and a minimal tool support needed. In Chapter 4, we successfully applied our process to recovery of the architecture and evolution of a non-trivial, industrial strength application.

Experience with the recovery of the architecture and evolution of XMMS has provided positive feedback and increased our confidence in the applicability and usefulness of the techniques that the process depends on. We found that a particular strength of the process is in the ability to perform recovery iteratively and incrementally. Also, the independence of particular techniques permits work distribution, which allows the whole development team to participate in the recovery process.

As our process is an enhancement of the original PBS approach, its main new strengths are the recovery of the business problem domain architecture and the emphasis on the recovery of the architectural rationale. In addition to these two, additional process advantages are:

- Minimal additional developer's training involved — all presented techniques are commonly used during forward engineering process, and are all commonly accepted and rec-

ommended practices in development community. This minimizes the additional training needed for the architecture recovery. Also, the simplicity of the presented techniques allows developers to quickly get up to speed in performing the recovery activities.

- Low risk incorporation into the development process — the risk of the incorporation of our recovery process into a development process is low as the cost of the tool support and developer training are low.
- Robust roundtrip tool support — as all minimal tool support (source code browsing and editing tool, drawing tool, text editor) already exists, and is of high quality, this goal is achieved.
- Reflects industrial best practices as much as possible — all mentioned activities are widely accepted in the software engineering community.
- Minimizes design activities and maximize programming — as all artifacts are minimal, and in a simple format they can be easily produced and maintained. Therefore, they do not present a large overhead while bringing benefits of systematic application of software architecture principles.
- Minimal and simple set of artifacts directly usable later for forward engineering and study of evolution — the simplicity of the produced artifacts and their interdependency allows them to be used for the further development of the system.
- Use of well established forward engineering principles to recover architecture rationale — our approach relies on use of quality attributes and forward design techniques to recover architecture rationale. Both are well studied and successfully applied in several other areas of software engineering.

The major obstacle during recovery was a lack of automated tool support for architectural use-case recovery. Even though we successfully used this particular technique in order to form a mental model of the dynamic interactions within the system, it required manual navigation through module interdependency references, which was very time consuming. As this technique

is very commonly used during actual development, and provides an efficient way for the documentation of high-level interactions within the system, a tool that supports automated use-case recovery should be developed.

Our future work will go in two different directions. The first one is further validation and refinement of our process through its application in development environments, and its adaptation for other development methodologies. The second direction is the development of new tools and improvement of existing tool support. Special emphasis will be given to development of a tool for automatic architectural use-case recovery and documentation. Also, a development of an integrated development environment for agile development processes will be considered, with particular emphasis on architecture recovery and refactoring components.

Bibliography

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, first edition, 1985.

[BBC+00] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi. The architecture based design method. *CMU/SEI*, 2000.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, first edition, 1998.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, first edition, 1999.

[BI96] B. Boehm and H. In. Identifying quality-requirement conflicts. *IEEE Software, Vol. 13, No. 2*, 1996.

[BMR+95] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Addison-Wesley, Reading, Massachusetts, first edition, 1995.

[Coc] Alistair Cockburn. Responsibility-based modeling. `members.aol.com/humansandt/techniques/responsibility.htm`; accessed December 1, 2001.

[Fow] Martin Fowler. The new methodology. `www.martinfowler.com/`; accessed December 1, 2001.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, first edition, 1999.

[FS99] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusetts, first edition, 1999.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. John Wiley & Son Ltd, first edition, 1996.

[GY] D. Gross and E. Yu. From non-functional requirements to design through patterns. `www.hkkk.fi/~mrossi/refsq/f113.pdf`; accessed December 1, 2001.

[IDE] Intellij idea. `http://www.intellij.com/`; accessed December 1, 2001.

[JAH00] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, Reading, Massachusetts, first edition, 2000.

[KKB+99] M. H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-based architecture styles. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 225-243*, 1999.

[KKC00] R. Kazman, M. Klein, and P. Clements. Atam: Method for architecture evaluation. *CMU/SEI*, 2000.

[Krua] Philippe Kruchten. The 4+1 view model of architecture. `www.rational.com/products/whitepapers/350.jsp`; accessed December 1, 2001.

[Krub] Philippe Kruchten. A rational development process. `www.rational.com/products/whitepapers/334.jsp`; accessed December 1, 2001.

[Lar01] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, second edition, 2001.

[MCY99] J. Mylopoulos, L. Cheung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of ACM, Vol. 42, No. 1*, 1999.

[PBS] Pbs. `swag.uwaterloo.ca/pbs/`; accessed December 1, 2001.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Ree] Jack W. Reeves. What is software design? `www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm`; accessed December 1, 2001.

[REF] Refactoring catalog. `www.refactoring.com/catalog/index.html`; accessed December 1, 2001.

[Rig] Rigi. `http://www.rigi.csc.uvic.ca/`; accessed December 1, 2001.

[Roy98] Walker Royce. *Software Project Management - A Unified Framework*. Addison-Wesley, Reading, Massachusetts, first edition, 1998.

[RR00] Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. Addison-Wesley, Reading, Massachusetts, first edition, 2000.

[RUP] Rational unified process (rup). `www.rational.com/products/rup/index.jsp`; accessed December 1, 2001.

[Sch98] Stephen R. Schach. *Classical and Object-Oriented Software Engineering With Uml and Java*. McGraw-Hill, fourth edition, 1998.

[SD96] Mary Shaw and Garlan David. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, first edition, 1996.

[SGG01] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, sixth edition, 2001.

[Shr] Shrimp. `http://www.csr.uvic.ca/shrimpviews/`; accessed December 1, 2001.

[TGLH00] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt. Architecture repair of open source software. *IWPC*, 2000.

[UND] Understand for c++. `www.scitools.com/ucpp.html`; accessed December 1, 2001.

[vLL00] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Vol. 26, No. 10*, 2000.

[Win] Winamp multimedia player. `www.winamp.com/`; accessed December 1, 2001.

[XMMa] X multimedia system (xmms). `www.xmms.org`; accessed December 1, 2001.

[XMMb] Xmms feature list. `www.xmms.org/features.html`; accessed December 1, 2001.

[XMMc] Xmms user manual. `www.xmms.org/documentation.html`; accessed December 1, 2001.