AGILE SOFTWARE ARCHITECTURE RECOVERY THROUGH EXISTING SOLUTIONS AND DESIGN PATTERNS

Chung-Horng Lung
Department of Systems and Computer Engineering
Carleton University
Ottawa, K1S 5B6, Canada
Email: chlung@sce.carleton.ca

ABSTRACT

Software architectures evolve over time due to requirement and technology changes. Hence, software architecture recovery is often necessary to capture and document existing systems to effectively support product evolution and maintenance. Architectures of existing systems can be recovered using reverse engineering techniques. Reverse engineering deals with deriving higher-level descriptions of a software system from existing software artifacts, primarily source code. Reverse engineering of source code, often, is a time consuming task. For reasons of limited resources or competition, software architectures could be recovered more efficiently by studying solutions from similar systems. This paper presents an approach for rapid and agile software architecture recovery in a mature domain, network applications. We demonstrate a case study for software architecture recovery by examining an existing architecture in the same domain. The existing architecture help derive a conceptual description for the target system. Meanwhile, some well-known design patterns in the similar domain are used to compare with the target system. The knowledge gained from design patterns provides more detailed information. The process is coupled with iterative reviews of the source code to refine the recovered software architecture.

KEY WORDS

software architecture recovery, reverse engineering, software evolution, design patterns, analogy

1. INTRODUCTION

Software architecture recovery is empirically important, because systems inevitably evolve over time due to changes in requirements, techniques, and personnel. Reverse engineering, hence, is a necessary means to capture a high level representation describing components and the structure of the components of the software. There have been numerous reports on reverse engineering in software engineering community, e.g. [20-22]. The recovered representation is critical in understanding the software. Understanding the software plays a crucial role to effectively maintain and evolve the software.

An existing system that has an up-to-date and complete documentation is rare in practice. Therefore, reverse engineering often is based on the source code, since it is the true description of the system. However, reading thousands or more lines of code to learn about various components and their relationships is very time-consuming. This is also true even with the aid of reverse engineering tools for complex applications. In addition, resources may be limited. Organizations may be reluctant to purchase a reverse engineering tool just for a short-duration usage. Furthermore, overhead may be involved in learning the tools, especially those tools that require re-compilation of the source code together with the tools. In such cases, based on practical experiences, just making the tool work with the source code may take several weeks even for a medium-size project.

On the other hand, it is not common to have a project that is totally new either in the problem domain or the solution domain. Indeed, there are new applications and advanced techniques. But many underlying logical concepts are similar to some existing systems, especially at the architecture level. Particularly, for mature domains, the concept usually is well studied and understood. Hence, some existing solutions could be used to facilitate the understanding of a new system.

This paper presents an agile approach to software architecture recovery (ASAR) based on the concept of analogy. Analogy is fundamental to reasoning. Analogy has been studied by many researchers in cognitive science, artificial intelligence, psychology, education, and philosophy [10]. Analogy is the process of transferring knowledge from an existing problem to a new problem that shares significant similarities – and using the transferred knowledge to construct solutions to the new problem [4]. The existing problem is often called base, while the new problem is referred to as target. The base is usually well understood.

Following a formal process in analogical reasoning is time-consuming. The approach presented in this paper, however, is rapid and agile in that it does not require a rigorous process as discussed in analogy. Analogy usually consists of four main steps: identification, retrieval, mapping, and evaluation. The main reason is that the effort for those steps can be eliminated or significantly reduced if the problem space is limited to a domain-specific area. The ASAR approach, therefore, advocates

adopting either the reference architecture or an architecture of a well-known or mature product in the same problem domain as the base. The base is compared against the target system. The source code of the target is then iteratively reviewed for understanding and refinement to reveal the actual software architecture. In addition, design patterns can be used together in the process. Design patterns provide recurring solutions and knowledge that can be reused in other applications.

The paper also presents a case study of the ASAR in network applications. The study consists of two systems, base and target. The target system had been developed to some point but later was suspended due to marketing reasons. The system became alive again several months later. But there were no documents for the target and the original designers had left. There was a need to recover the architecture of the system for new designers in a short period of time. However, the available resources, both time and human, were limited. The ASAR approach meets the requirements and effectively supports the recovery and revolution of the target.

The intent of the paper is not to downgrade the value of the conventional reverse engineering effort from source code. Rather, the main purpose of this effort is to present an alternative to support rapid and agile software architecture recovery under the constraints of limited resources and demands of time-to-market. Moreover, the goal is to recover the architecture approximately instead of exactly to help new designers better understand the system and start working on the system.

The rest of this paper is organized as follows. Section 2 briefly describes the target system. Section 3 presents a base architecture obtained from a conceptually similar product and illustrates the recovery of the software architecture of the target system by using the existing base architecture and some well-known design patterns. Section 3 also describes a generalized form derived from the base and the target. Section 4 discusses some related work. Finally, section 5 is the conclusion.

2. BACKGROUND OF THE TARGET PROBLEM

The target system is for network traffic engineering based on CR-LDP (Constraint-based Routing Label Distribution Protocols) and MPLS (Multiprotocol Label Switching) protocols [5]. MPLS is regarded as a fundamentally important technology for the next-generation internet.

The software is not very large in size. It consists of about 30,000 lines of code. However, the software contains complicated operations, algorithms, and domain knowledge in networks and traffic management. The bulk of the software was primarily written in a short time to showcase in an international convention. Some parts of the software were better written in object-oriented style, but many parts still followed the C style. Figure 1 shows the high level view of the design derived from the executable software processes.

As shown in Figure 1, the router processes are symmetrical and identical. The four router processes depicted in Figure 1 are the same, except possibly that the number of connections to other routers may be different. Those router processes can be run on the same machine concurrently or on separate machines. A generator process randomly generates data packets that will be forwarded to other router processes and a sink process consumes data packets received from other router.

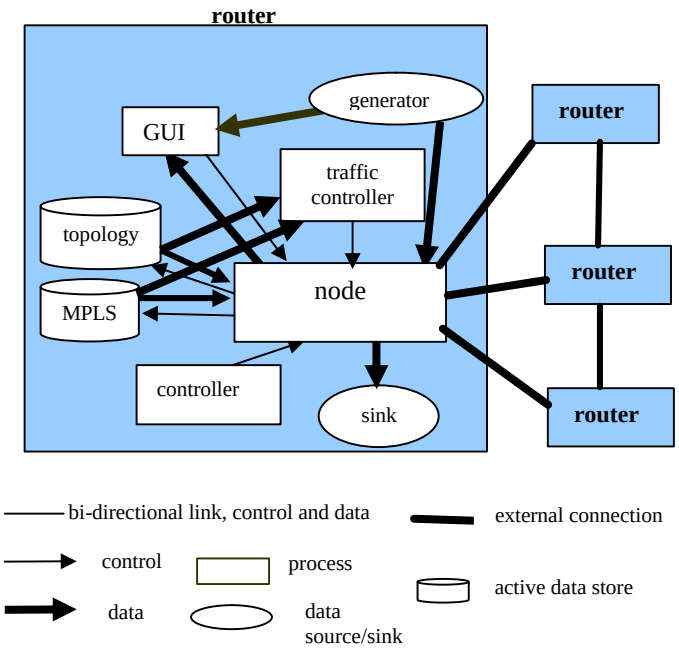

Figure 1. Software Process View of the Target System

There was no design documentation for the software, except the user's guide. Although the project is not very large, it is still challenging. A person's "span of understanding" of software is only 7000 to 15,000 lines of code; and to understand this amount of code requires about three to six months of time [18]. There was a need to capture the architecture in a limited time without any reverse engineering tools or many resources available.

The architecture recovery process started with reading the directory structure of the target system. But it didn't go far. All the files actually were in one directory. And the size of the files or classes varied dramatically, raging from just several lines of code to over ten thousand lines of code. Instead of browsing through the source code, we turned the effort to look at a conceptually similar and mature product in the same problem family to obtain a mental model. The mental model is used to understand the main components, their relationships, and applications. The next section describes the approach.

3. AGILE SOFTWARE ARCHITECTURE RECOVERY APPROACH

In analogy community, researchers emphasize three types of information: syntactic, semantic, and pragmatic. Maiden and Sutcliffe [17,24] and Lung, et al [14-16] apply some crucial ideas in analogy to software reuse and domain analysis. Those papers [14-17,24] and articles in analogy community show that analogy may support knowledge transfer across problem domains. The main idea of those techniques is to identify main components, their relations, major functionalities, higher-order relations (relations of relations), and system dynamics. The artifacts are then used for identifying a similar base problem and mapping to the target problem.

The ASAR approach is based on the concept of analogy, but is simplified. ASAR is an informal and iterative process consisting of the following steps:

1. Retrieve the reference architecture or an existing architecture from another product in the same or similar problem domain to serve as the base.
2. Evaluate and understand the base system.
3. Identify key components and relations in the target system.
4. Map the structure of the base to the target system to obtain a conceptual model for the target.
5. Compare design patterns with the target system to fill in the gap between the conceptual model and the source code, and refine the architecture of the target system.

The following paragraphs discuss each step in more detail. First, we start with examining systems in a similar domain. The effort of identifying an analogous problem and mapping of knowledge from the base to the target can be reduced. The reason is that for similar or mature domains, main components, their relations, major functionalities are well understood and similar. For mature domains, design patterns also have been extensively studied and documented [9].

Evaluating and understanding the base is necessary before mapping the architecture to the target. The reason is trivial. This step can be skipped if the analyst is already familiar with the base. Another step that is useful before mapping is to identify the key components in the target. The key components and their relations are then used to compare with those of the base. This step usually can establish partial mapping between the two problems.

Based on the knowledge of the base system and the partial mapping of components and relationships, the third step deals with further mapping between the two problems. The emphasis is on the structure of the problems. The idea follows the structure-mapping principle [8], which is considered as one of the most important analogy theories [10]. Structure-mapping depicts that analogy is a mapping of systems of relations governed by higher-order relations. A high-order relation captures the relation of relations rather than relations of components. Some isolated features usually are not adopted for mapping.

Design patterns also can serve as reference entities. The idea is similar to that of reference architectures, but the scope is smaller. Here we assume that some design concepts of the problem are similar to that of some design patterns. The theory is that the domain is stable and those patterns are identified as a result of repeated usages. Similar to step 1, the patterns are used to help understand the target. The purpose of this step is not the other way around, meaning detecting the patterns in the target system. Knowing some patterns and then reviewing the source code from the perspective of patterns facilitates better understanding of the target system.

3.1 The Architecture of the Base

The target software is an application of network traffic engineering. It is a very new area – new protocols, new switched techniques, and new traffic engineering concepts. On the other hand, the system was not totally new from a higher-level perspective. If we hide some technology details, the system actually shares some critical commonalities with other network systems.

Network software has been developed and studied extensively. Many solid conceptual foundations have been captured and documented. This motivates us to start with looking around to find a mature product and then to read relevant design patterns [23] from the open literature for better understanding of network computing.

Despite the sheer diversity of networks, protocols, standards, and applications, there are common conceptual similarities. We know the application area and the relevant protocols of the target. So, we can abstract out those specific areas and identify generic features to help us understand the overall structure of a network system.

A subsystem of the product that is related to the target is used as the base for understanding and comparisons. The base was developed by a different product group in a large company and has been used for several years. The base was well documented via a thorough reverse engineering effort. The reverse engineering involved three people (one designer and two co-op students) without much prior knowledge of the system and the problem domain for four to five months. Designers of the base were solicited for support if necessary during the process.

The main objective is to help us understand the target system by learning from a mature product. The base system is much larger than the target system. It has more than 100 KLOC. Figure 2 highlights the architecture, including the main components and their interconnections.

Bejar et al. [2] postulate that both similarity and difference are crucial in analogical reasoning. We follow the idea to examine both similarities and differences between the based and the target. The base system in fact is very different from the target system in many aspects if we get into more details. Here are some examples. The first obvious difference is in the area of protocols. The

target system uses CR-LDP and MPLS, which do not exist in the base. The base contains a Protocol Framework that is designed to inter-work with a number of standards and network protocols. However, those protocols are very different from CR-LSP and MPLS that the target system is dealing with. Secondly, the base has a very high demand in performance and throughput. Hence, shared memory is adopted as the inter-process communications mechanism to improve performance. As a result, the base has complicated synchronization methods between software processes. In addition, the base has a gigantic third party real-time database management system (RDBMS), which contains millions of subscriber records, and there are sophisticated operations associated with the RDBMS.

Nevertheless, the abstraction of the base still serves as a useful vehicle to help understand the basic concept and devise the overall structure. For our purpose, the complicated shared memory and database are not our main concerns, because the target system doesn't have a real database and the inter-process communication method is primarily sockets. These two areas are not difficult to find in the target system. Hence, we can hide the relevant detailed design and implementation information. Another example is the protocol framework and the service framework shown in Figure 2. As stated earlier, the target system uses completely different protocols and services. But again, the abstraction layer is similar because the target system also needs to deal with protocols and services. The Protocol Framework in the base system also demonstrates how it interacts with other components, particularly Message Processor and Request Handler. The interrelationships among those components are valuable in understanding the target system.

Another source that helps us better understand the target system is design patterns. Many design patterns have been identified and documented; including the area of distributed and network computing [Schmidt00]. We assume that the target system, although not written based on design patterns, contains some similar concepts, since it belongs to the same problem domain and the domain is stable. Thus, the next step is to compare some design patterns with the target system.

The approach does not suggest reading the code and identifying various patterns. Detecting patterns in existing software may not be trivial and usually is time consuming. Rather, we choose design patterns first in some specific areas and then review the code based on the concept of those patterns. For this study, we start with patterns for concurrent and networked objects. That way, we have more concrete goals derived from specific patterns to look for and the person performing the analysis does not need to be very experienced in patterns. It is not difficult to identify some areas in the target system that shares commonalities with patterns because the target is in the area of communications.

For example, the Wrapper Façade pattern deals with the communications setup. We then check the source code to see how this part was realized in the target. The pattern is close to the way the connections are set up in the target system. Similarly, the dispatch service in the target system is relevant to the Reactor pattern that integrates the synchronous demultiplexing of events and the dispatching of their corresponding event handlers. The Acceptor-Connector pattern is almost identical to the way the routers connect to each other during system startup.

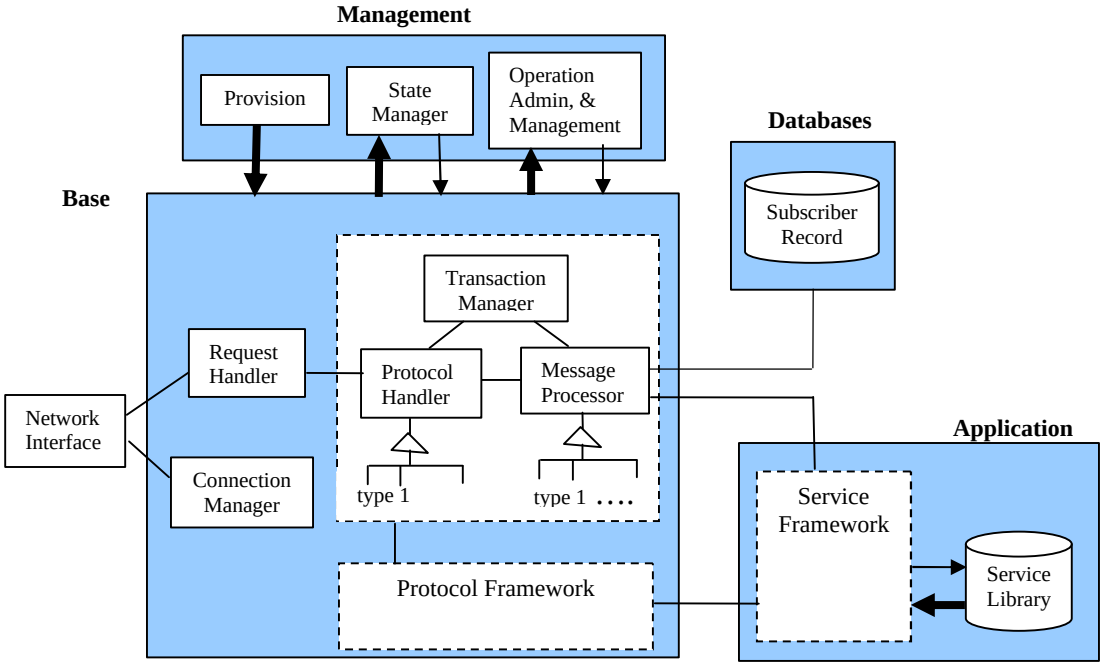

Figure 2. Software Architecture of the Base Problem

For some areas, it takes deeper analysis and code review to identify patterns that serve similar purposes. An example is the relationship between the node process and the traffic controller process illustrated in Figure 1. The traffic controller may send commands or queries to node. When the traffic controller invokes asynchronous commands on the node process, it stores information that later will be used to identify the command's completion. This is similar to the Asynchronous Completion Token.

The target system is designed with multiple treads, which requires synchronization techniques in concurrent processing. There are several relevant patterns for concurrent processing and fundamental synchronization. Among them, Active Object pattern and Half-Sync/Half-Async patterns are relevant and provide useful insights for further understanding the design.

For instance, the Active Object design pattern decouples method execution from method invocation to improve concurrency and simplify synchronization. The idea is also adopted in the target system. There is a thread that handles method execution based on the incoming messages from other routers. Multiple threads are used to send outgoing messages to various destinations. In addition, the traffic controller process is also tightly coupled with the Strategy pattern [7]. The traffic controller process was designed to encompass various algorithms for load balancing and route calculation. The Strategy pattern fits the objective very well.

3.2 Recovered Software Architecture

By studying an existing architecture, reviewing well known relevant design patterns in the same area, and iteratively checking related source code, the architecture of the target is recovered in 2-3 weeks by one person without rigorous software reverse engineering tools and process. The person was knowledgeable of the base, but was not familiar with the relevant patterns at that time. The recovered architecture, as shown in Figure 3, is then used for the new designers to continually work on the system. The node class is conceptually decoupled into several more independent units, which paves the way for future re-engineering.

As presented in Figure 3, the basic structure of the target is similar to the base system. There are four main components: *node, management, data stores, and application*. With the exception of *data stores*, the other three units are connected using TCP or UDP socket utilities. In other words, those components talk to each other through network messages. There are multiple instances of node, each running on a separate thread. Those threads share data stores, stats, and queues (not shown in the diagram) connecting to the Message Processor. The Request Handler is closely related to the Reactor design pattern. The connections are primarily handled by routines grouped in the Connection Manager. Connection Manager is similar to the Acceptor-Connector. Another routine, utility (not shown in the diagram) provides the wrapper-like function as described in the Wrapper Façade pattern.

The Protocol Handler is much simpler than the one implemented in the base problem, since there is only one protocol adopted for the problem. After a message is decoded, the Message Processor invokes appropriate handlers to perform the action. Likewise, messages could come from those handlers to the Message Processor and be encoded by the Protocol Handler, and then be sent to the network.

The traffic controller class contains several sub-classes for various platforms. The traffic controller also was designed to accommodate different algorithms for traffic management. As stated earlier, this part is directly related to the Strategy design pattern.

3.3 A Generalized Form of Network Applications

A generalized architecture for network applications can be derived from the two cases studied. The idea is similar to the reference architecture. Modifications may be needed if more cases are studied.

The generalized form consists of the following main components:

Connection Manager (CM)**.** The CM deals with the connections setup, teardown with other network elements. Messages are transmitted through the CM.

Request Handler (RH). When a router receives a message from other network elements, the message actually is a request which will be handled by the RH.

Protocol Handler (PH). PH encodes and decodes messages.

Message Processor (MP). After a message is decoded by the PH, MP reacts to the message. MP interacts closely with the PH.

Transaction Manager (TM). TM is used to keep track of transactions. A transaction may involve multiple messages.

Data Base (DB). One or more DBs usually are required for information lookups and updates.

Services or Applications. This is the application layer. Once the message is properly processed, this layer provides actual services.

System Management (SM). This component mainly deals with provision, operation, and administration supports.

4. Related Work

Obviously, the paper is directly related to reverse engineering. Numerous papers have been published in this area [20-22]. Most of these papers deal with the transformation from a lower-level representation, usually source code, to a higher-level abstraction. As depicted in the introduction, this approach is appropriate in many cases where resources are not an inhibiting factor. Here, we will not discuss further in this topic. Rather, other disciplines that share similar ideas are briefly mentioned.

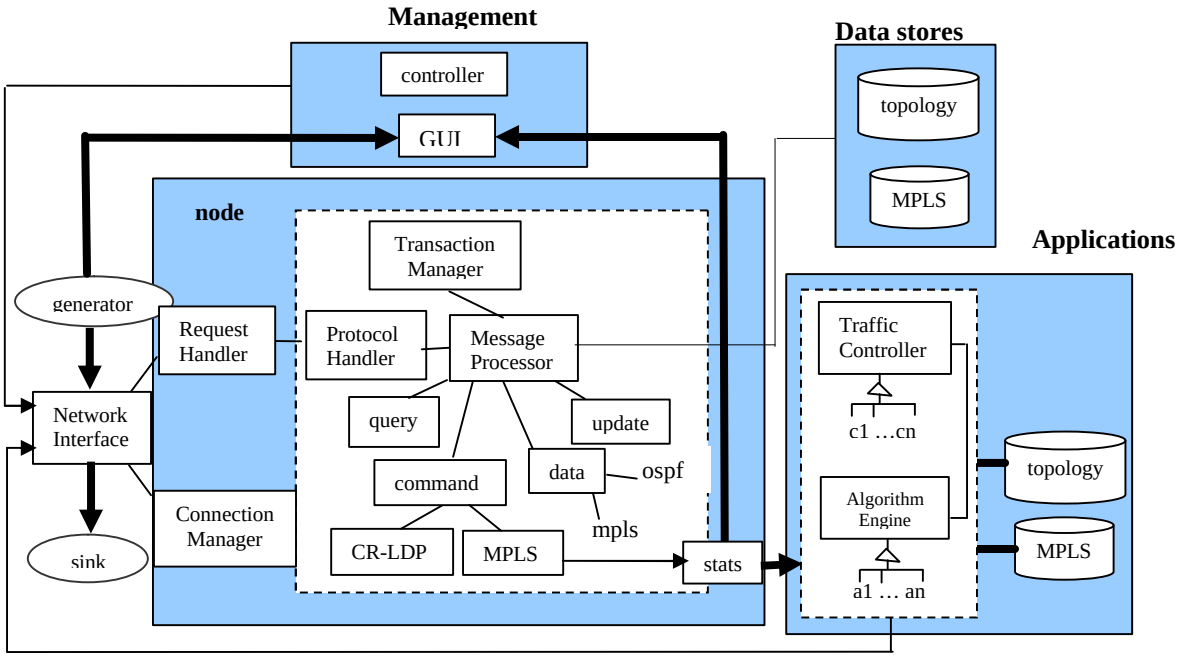

Figure 3. Recovered Software Architecture

This main idea of this paper is similar to the concept of the reference architecture [3,9]. The reference architecture for a domain defines the fundamental components of the domain and their relations. The reference architecture enables reuse, reduces development cost, and improves communications among various groups. This paper does not address reference architecture specifically. However, for mature domains, a base system could be used as the "reference architecture".

This study is also similar to the concept of case-based reasoning. Case-based reasoning means reasoning based on previous cases or experiences. Kolodner and Leake [12] point out that case-based reasoning provides a wide range of advantages. Two of them are directly relevant to this article:

- Case-based reasoning helps propose solutions quickly rather than derive the answers from scratch.
- Case-based reasoning helps propose solutions in domains that aren't well understood.

Another related topic is analogy. As stated in section 1, analogy has been studied in cognitive science, artificial intelligence, psychology, and philosophy [2,10]. Analogy is the process of transferring knowledge from an existing problem to a new problem that shares significant similarities – and using the transferred knowledge to construct solutions to the target problem. Analogy has also been applied to software engineering [14-17,24]. Readers can refer to these papers for details.

Knowledge management is also relevant to this area [25]. Knowledge management addresses the way of capturing, organizing, retrieving, using, and learning the knowledge. The Experience Factory [1] is an infrastructure for organizational reuse and learning. One issue is the identification and retrieval of a similar system.

Identification of reusable components is also an issue that has been discussed in software reuse [13] or domain engineering [19]. In general, this may be a difficult question and may involve technical, organizational, and management issues. Furthermore, there may be high cost overhead. Although this area has been intensively investigated, formal and systematic reuse in practice is difficult to achieve and often is ineffective [6].

Our case study dealt with a simpler problem. A base case in the similar application domain was selected to facilitate this exercise. How do we know which system to choose as the base? In theory, this is a difficult. In our case, it may seem opportunistic. In practice, however, a partial mapping between the base and the target is necessary for the retrieval of the base. The partial mapping is primarily based on problem domain and domain knowledge. The case study was performed on a mature domain. This was the main reason that the base problem was selected. Domain knowledge is one of the most important success factors in reuse and knowledge transfer.

5. Conclusion

This paper described an approach to software architecture recovery based on an existing product in the same area and design patterns under the resource constraint. The main objective was to get a mental structure of the system, and to support the new designers to understand and evolve the system in a short period of time. The approach first selects the reference architecture or an existing system in the same problem domain to serve as the base. The base system usually is well known and is used to support understanding of the target problem. Iteratively, the base is compared with the target

to identify similarities and differences. The derived knowledge can then be tailored to construct a high-level software architecture of the target system. Design patterns provide more detailed information, which fills in the gap between the conceptual architecture and the source code.

The approach was not meant to replace the conventional reverse engineering wisdom for software architecture recovery; instead, the article presented an empirical study and an alternative way for rapid architecture recovery.

The approach was successful in terms of meeting our objectives – rapid architecture recovery and capture of critical design knowledge. There are some factors for the success. First, the problem is in a mature domain. In other words, the target system shares many domain specific features with other similar systems. The fact that some areas of the system are closely related to the concept of some design patterns also speaks for this fact. This point also concurs with the belief that domain analysis is more effective in a well-scoped and stable application domain.

Next, a system, which shares similar high-level features, is available to server as the base. The base system is well understood and documented. There was little search time needed for the base problem. The reference architecture, if available, also can be used for this purpose. Retrieval of the base, in general, is a difficult task. The problem can be simplified, as advocated in this paper, by focusing on the same or similar problem domain.

Another point is that the size of the base is larger than that of the target. In this case, the base contains more features that help understand the target. If the base is smaller and simpler, it may not provide as valuable information or knowledge to comprehend a more complex target system. Thirdly, the size of the target is small. On the other hand, a large system may be divided into smaller sub-systems for design recovery.

On the other hand, this approach might not work well for an application where similar solutions or the reference architecture are difficult to find or do not exist. Another assumption for this approach is that the base case is relatively well understood. If not, even if the base and the target are very similar, the approach may not reduce or simplify the reverse engineering process and time.

Another issue is what if the base chosen does not match well with the target. This is still a research area in analogy and domain analysis. This is beyond the scope of this paper. Classification of domain models [14], facet classification scheme, and library-based approach [19] have been proposed. However, cost overhead and ineffectiveness are still main barriers [6] for those approaches. This paper, on the other hand, advocates an approach based on better-known applications, which in general will be more effective.

REFERENCES:

[1] V.R. Basili, G. Caldiera, and H. D. Rombach, Experience Factory, in J. J. Marciniak, ed., *Encyclopedia of Software Engineering*, vol. 1, John Wiley & Sons, 1994, 469-476.

[2] I.I. Bejar, R. Chaffin, S. Embretson, *Cognitive and Psychometric analysis of analogical problem solving*, Springer-Verlag, 1991.

[3] J. Bergey, G. Campbell, P. Clements, S. Cohen, L. Jones, R. Krut, L. Northrop, and D. Smith, Second DoD *Product Line Practice Workshop Report*. Technical Report CMU/SEI-99-TR-015, Carnegie Mellon University, Oct. 1999.

[4] J.G. Carbonell, Learning by Analogy: Formalizing and Generalising Plans from Past Experience, in *Machine Learning: An Artificial Intelligence Approach*, Springer-Verlag, 1983.

[5] B. Davie and Y. Rekhter, *MPLS Technology and Applications*, Morgan Kaufmann Publishers, 2000.

[6] R.G. Fichman and G.F. Kemerer, Incentive Compatibility and Systematic Software Reuse, *J. of Sys & Sw*, 57, 2001, 45-60.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[8] D. Gentner, Structure-Mapping: a Theoretical Framework for Analogy, *Cognitive Science 7(2)*, 1983, 155-170.

[9] A.E. Hassan and R. Holt, A Reference Architecture for Web Servers, *Proc. of the Working Conf. on Reverse Eng*, 2000.

[10] D. Helman, ed., *Analogical Reasoning*, Kluwer Academic Publishers, 1988.

[11] R. Holt, Software Architecture as a Shared Mental Model, *Proc. of the Int'l Workshop on Program Comprehension*, 2002.

[12] J. L. Kolodner and D. B. Leake, A Tutorial Introduction to Case-Base Reasoning, in *Case-Base Reasoning Experiences, Lessons, and Future Directions*, The MIT Press, 1996, 31-66.

[13] C.W. Krueger, Software Reuse, *ACM Computing Surveys 24(2)*, 1992, 131-183.

[14] C.-H. Lung and J.E. Urban, An Approach to the Classification of Domain Models in Support of Analogical Reuse, *Proc. of Symp. on Software Reusability*, 1994, 169-178.

[15] C.-H. Lung and J.E. Urban, An Expanded View of Domain Modeling for Software Analogy, *Proc. of Int'l Computer Software & Applications Conf*, 1995, 77-82.

[16] C.-H. Lung, G. Mackulak, and J. Urban, Software Reuse and Knowledge Transfer through Analogy and Design Patterns, *Proc. of the Int'l Con. on Sw Eng Research & Practice*, 2002.

[17] N.A.M. Maiden and A.G. Sutcliffe, Exploiting Reusable Specifications through Analogy, *Commu. of the ACM 35(4)*, 55-64, April 1992.

[18] C. McClure, The three Rs of software automation: re-engineering, repository,reusability, Prentice Hall, 1992.

[19] R. Prieto-Diaz and G. Arango, Introduction and Overview: Domain Analysis Concepts and Research Directions, in *Domain Analysis and Software System Modeling*, IEEE Computer Society Press, 1991, 9-32.

[20] *Proc. of the International Conf. on Software Maintenance.*

[21] *Proc. of the Int'l Workshop on Program Comprehension*.

[22] *Proc. of the Working Conference on Reverse Engineering*.

[23] D. Schmidt, M. Stal, H. Rohnert, and R. Buschmann, *Pattern-Oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[24] A.G. Sutcliffe and N.A.M. Maiden, Analogical Retrieval in Reuse Oriented Requirements Engineering, *Sw Eng J.*, 9, 1996.

[25] K.M. Wijg, Knowledge Management: Where did it come from and where will it go?, *Expert Sys with Applications 13*, 1997, 1-14.