CIS Ph.D. Comprehensive Examinations
Research Readiness Question
Improvement Research Proposal

Maintenance and Evolution Patterns: Automated Detection and Retrieval

Based on:

Kannan Mohan
Department of Computer Information Systems
Georgia State University
August 20th, 2001
1 Introduction and Motivation

Software maintenance demands a large portion of organizational resources, with expenditures running as high as 50-80% of Information Systems budget (Parikh 1982; Nosek and Palvia 1990). (Brooks 1982) claims that the total cost of maintaining a widely used program is 40% or more of the cost of developing the same. It is also noted that more than 50% of programmer effort is dedicated to software maintenance (Gibson and Senn 1989). Experts in the field indicate software maintenance as one of the major challenges of the past decade (Yourdon 1992). (Coleman, Ash et al. 1994) shows evidences from practice endorsing the need to improve the process of software maintenance. This phase of software engineering is inundated with problems due to a variety of reasons. Some of the reasons widely quoted in the literature are (Kemerer 1995): (a) Poor quality of older existing software, (b) Systems are not designed with maintenance in mind, (c) Maintenance does not attract the best software talent, (c) Lack of proper structure in the design of existing software, and (d) Lack of documentation of the history of design.

Past research has attempted to alleviate these problems by addressing the issues that cause these problems at various stages of software development life cycle. (Gamma, Helm et al. 1995) address the problem of redesign as a result of change using design patterns. (Shaw and Garlan 1996) propose an architectural approach to software so as to enhance the capability to handle size and complexity. (Ramesh and Dhar 1992) suggest a conceptual model to capture the rationale behind design decisions that would help maintenance personnel understand interactions among various parts of the software. Techniques like structured design and object-oriented design facilitate change by organizing and localizing information processing in an application (Pfleeger 1998). (Banker and Slaughter 2000) investigates the role of structure in reducing enhancement costs and errors. But in the case of most legacy applications, such approaches cannot be applied, as these systems have already been developed and are in production. Most of these systems are infested with the problems listed above that proliferates the effort required for maintaining these systems. As opposed to process improvements in the early phases of the software development life cycle, the scenario for legacy applications require improvements in the maintenance phase as they have already gone through the earlier phases. This research focuses on alleviating the above-described problems through the usage of maintenance and evolution patterns.
Prior research on software engineering focuses primarily on addressing problems in the early part of the life cycle so as to reduce cost and effort in the later phases. They tend to suggest solutions that can be applied to requirements analysis and design phases of the life cycle so as to reduce maintenance effort. (Rugaber 1999) discusses the tool support provided for software evolution through software reengineering. His effort focused on addressing problems in software evolution due to new requirements. (Wu, Pan et al. 2000) present a technique that could be applied to the various maintenance activities, but the technique is restricted to component-based software. Few empirical studies have investigated the reasons for changes and portray the current software maintenance practices in the industry (Singer 1998). They discuss the role of maintenance control systems in aiding maintenance personnel. Bug tracking database is used as a repository of information about errors corrected in the past and maintenance personnel try to find solutions to current problems by searching the database to identify similar problems solved earlier. But the systems as such do not suggest solutions by matching patterns of errors. This research focuses on automating the process of identifying, abstracting and storing maintenance and evolution activities in the form of patterns, and later retrieving the same to suggest solutions to problems similar to those that have been previously solved.

2 Research Objectives

This research focuses on identifying and abstracting the activities performed during maintenance and evolution of software systems and recording them so that they could be used to address similar problems that might arise in the future. Maintenance and evolution patterns that are identified form the templates for abstraction. Since maintenance and evolution activities could lead to changes in artifacts that had been generated during any phase in the software development life cycle, our objective is to capture such activities associated with any artifact in the form of patterns and abstract the same. These objectives can be represented as research questions as follows:

How can maintenance and evolution patterns be leveraged to enhance the process of software maintenance and evolution? This question can be further granulated as follows:

- How can we capture maintenance and evolution patterns?
- How can the captured patterns be abstracted and rendered context-free?
- How could these patterns be stored in a repository so as to facilitate effective retrieval when required?
• How can we match current problems and solutions with those that are available in the repository thereby providing effective suggestions to maintenance personnel?

3 Theoretical Background

3.1 Definitions: Legacy Software, Maintenance and Evolution

Legacy applications are those that have been handed down from a previous generation of developers (Noffsinger, Nidebalski et al. 1998). (Liu, Alderson et al. 1999) defines ‘legacy’ systems as those that have become inadequate in terms of capacity and functionality. These applications go through heavy changes and adaptation so as to suit to the new functional and technological requirements. These systems are the ones that demand and are subjected to heavy maintenance.

(Canning 1972) reported that practitioners describe maintenance narrowly as correcting errors, and broadly as extending and expanding software functionality. Swanson’s (Swanson 1976) typology was based on the system owners’ intention for requesting maintenance. He pointed out to three intentions for undertaking maintenance: (a) to perfect the system in terms of performance, processing efficiency or maintainability (perfective maintenance), (b) to adapt the system to the changes in its environment (adaptive maintenance), and (c) to correct processing, performance or implementation failures of the system (corrective maintenance) (Chapin 2000). Later, IEEE formalized the definition of software maintenance consistent with Swanson’s definition (ANSI/IEEE 1983).

Software evolution refers to the dynamic behavior of programming systems as they are maintained and enhanced over their lifetimes (Belady and Lehman 1976). Rugaber (Rugaber 1999) defines software evolution as the process of adapting an existing software system to conform to an enhanced set of requirements. Kemerer et al. (Kemerer and Slaughter 1999) delineate the distinction between maintenance and evolution. ‘While maintenance refers to the activities that take place at any time after the project is implemented, evolution refers to the dynamic behavior of systems and how they change over time’. (Hsi and Potts 2000) describes ‘feature creep’ as a phenomenon of software evolution. Apart from growing in size and complexity, the system also shows a reduction in conceptual homogeneity or intellectual coherence of the product as experienced by the user. Evolution could be described from various perspectives. Architectural evolution occurs when the architecture of the system changes, thereby making the system more adaptable. When the developers predict the need for a variety
of functionalities to be added and add those in forthcoming versions, the system is said experience functional evolution.

3.2 Maintenance, Evolution and Patterns

(Burch and Kung 1997) describes four distinct maintenance stages. Users learn during the first stage, training being offered by the IS department. As they get used to the system, they find bugs in the system, which leads to the second stage where detected errors are corrected. During the third stage, the system stabilizes and the users demand more features. Finally, during the last stage, users ask for more features that could not be supported by the old technology and hence they choose to replace the old system with a new one (Burch and Kung 1997). During each stage, maintenance personnel would be facing problems identical in nature, repeating over time. Kemerer et al. (Kemerer and Slaughter 1997) have conducted a longitudinal study in order to investigate the patterns of software evolution and to identify specific drivers of maintenance patterns. If maintenance personnel have access to an effective system that acts as a repository of the past problems and solutions, they can avoid solving identical problems each time from the scratch. These recurring principles and idiomatic solutions help software engineers reuse successful solutions by basing new solutions on prior experience. Such principles and idioms, if codified in a structured format describing the problem and solution, and given a name, can be called as a pattern (Larman 1998).

Christopher Alexander, who introduced the formal notion of patterns in architecture (Alexander, Ishikawa et al. 1977; Alexander 1979), says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”. This concept of patterns has been used in various fields, including software engineering. Designers and developers use patterns as a shared language describing problem/solution pairs in various stages of the software development life cycle. (Saeki 1999) discusses the abstraction of requirements descriptions written as use cases into use case patterns, so that analysts can reuse the experiences in requirements analysis. (Coad 1992) enumerates and describes a list of patterns that could be used in the analysis stage of the software development. (Gamma, Helm et al. 1995) classify and explain patterns that could be used during the design phase.
3.3 Pattern identification, retrieval and usage

The above section delineated the widespread usage of patterns in various stages of software development life cycle. But the usage of patterns during the maintenance stage has not been extensively studied with the only exception of the longitudinal study that had been conducted by (Kemerer and Slaughter 1997). This study’s primary focus is on empirical investigation of software evolution patterns and the drivers of maintenance patterns, not on automated identification, retrieval and usage of the same. Purao et al. (Purao and Storey 1997) have developed a methodology for intelligent retrieval and synthesis of patterns so as to automate the design process. But this research is based on the assumption that libraries of patterns are available. None of these studies have attempted to automatically identify maintenance patterns, store them in a repository and retrieve relevant ones that are identical to the problem at hand.

4 Identification and Retrieval of Maintenance Patterns

A tool, imparted with the capability of identifying, storing and retrieving maintenance patterns is integrated with the interface provided to the maintenance personnel for making changes to artifacts produced during any phase of the software development life cycle. Figure 1 depicts the proposed architecture, which shows the various components of the pattern recognition and retrieval system (called MPRRS hereafter – Maintenance Pattern Recognition and Retrieval System) and how it interacts with the interface provided for the maintenance personnel to modify the legacy system. Analyzer, pattern base, tracer and the abstracter are the components that are part of the MPRRS. The arrows among the various components shown in Figure 1 indicate the direction of flow of information. The following subsections describe the roles played by each component shown in Figure 1.

4.1 Legacy system

The legacy system shown in the architecture depicts the environment in which maintenance personnel work. The interface would include any CASE (Computer-Aided Software Engineering) tool that is used to modify use cases and design models, integrated development environments, used to modify code fragments and any other tool, used to modify any artifact produced during any phase of the development life cycle, provided it supports customization.
A major problem associated with maintenance is that in most cases it is done at the code level. As the system passes through a few cycles of maintenance, the other related artifacts become obsolete in the sense that they are not changed in order to be synchronized with the code. For instance, if a change request is received, which actually demands a change in one of the requirements, an ideal way to handle this would be to start the modification at the level of requirements and then proceed downwards following that requirement’s trace through other artifacts. But this is rarely done in practice. Code is the only artifact that is changed leaving the other artifacts inconsistent with the code. Here we assume that the maintenance personnel can modify any artifact associated with the application.

4.2 Analyzer

When the maintenance personnel initiate a change in any fragment of an artifact, the analyzer studies the nature of the change in terms of various parameters, compares it with those of the patterns available in the pattern base and provides a suggestion to the user about similar situations that had been solved earlier. The user should be able to traverse to these fragments that were changed earlier using the tracer. (Kontogiannis 1997) has developed a fast technique to identify programming patterns based on five software metrics. These software metrics are calculated based on certain features of the software. Distance metric used, is the Euclidean distance between elements on a five dimensional space created by the software metrics.
(Mayrand, Leblanc et al. 1996) discusses a similar technique used to identify function cloning. Function metrics have been used to compare functions and identify the level of cloning, the cloning scale ranging from ‘distinct’ to ‘exact copy’. Similar techniques can be applied here for identifying patterns in code. But these techniques have to be modified to acquire the capability to identify patterns in any artifact fragment. Key features have to be identified for each type of artifact and these features should be used to calculate related metric values. Depending on the value of the parameters used to identify patterns, the analyzer could further compare artifacts associated with the one that was compared in order to identify patterns of change at a different level. These related artifacts can be retrieved with the help of the tracer. Type of evolution and evolution distance are important parameters that provide a bearing on the nature of evolution (Tahvildari, Gregory et al. 1999). Tahvildari et al. (Tahvildari, Gregory et al. 1999) discuss three types of evolution: Interface, implementation and structural evolution, and they explain techniques to calculate evolution distances for each of these types of evolution. MPRRS has to interact with the maintenance and version control system so as to get the required inputs to calculate these distances. Architectural recovery techniques, (Eixelsberger, Ogris et al. 1998) used to recover architectural properties and descriptions of a legacy system, could be used by the analyzer to recover valuable information from the legacy system.

4.3 Tracer

Instead of storing extensive details about the modified artifact fragments in the pattern base, MPRRS stores only important parameter details. But the analyzer needs to compare artifact fragments for new cases in order to calculate the various metric values. This necessitates the need for a means to access artifact fragments. The tracer could be a requirements traceability tool, modified to fit in this architecture and that which has the capability to maintain traceability among the various artifacts. It helps the analyzer in accessing various artifact fragments that are associated with specific patterns.

4.4 Abstracter

The abstracter works in conjunction with the analyzer to generalize the patterns found. Applicability is increased by converting contextual patterns to context-free patterns. As one person’s pattern could be another person’s primitive building block (Gamma, Helm et al. 1995), both contextual and context-free patterns are stored in the pattern base.
4.5 **Pattern base**

The pattern base acts as a repository of maintenance and evolution patterns. Various parameters, metric values used to calculate distance and trace information that is to be used by the tracer are stored in the pattern base. Apart from storing patterns that are automatically identified, user-specified patterns could also be stored in the pattern base. Patterns at various levels of granularity, contextual and domain dependent patterns can be stored in the pattern base.

5 **Evaluation**

Building artifacts and evaluating the same to test for feasibility are the two research activities in design science (March and Smith 1995). This research can be evaluated for its effectiveness by conducting experiments using the artifact that is developed. Sample maintenance scenarios can be created for subjects who would be software maintainers, and recall and precision of patterns that are identified, abstracted and stored by MPRRS can be compared to those for the patterns that an observing expert could identify. The two metrics, recall and precision would measure retrieval quality (Purao and Storey 1997). We share the same definition as Purao et al. Recall is defined as the fraction of relevant patterns retrieved, while precision is the fraction of retrieved patterns that are relevant. The measures used for synthesis quality, viz., coverage and spuriousness, by (Purao and Storey 1997) are adapted and used as metrics for identification quality. Coverage is defined as the fraction of patterns identified, while spuriousness is defined as the number of patterns incorrectly identified. Distance metric could be used to evaluate the closeness of the patterns retrieved to the current scenario. Again, this metric can be calculated for MPRRS retrieved patterns and patterns retrieved by human experts. This is a form of Turing’s test where we compare machine performance with human performance. A system is said to have passed the Turing’s test if its responses are indistinguishable from that of a human being. In this case, we would compare the effectiveness of MPRRS in identifying, abstracting and retrieving patterns with that of a set of human experts, based on metrics like precision, recall, coverage, spuriousness and distance. Further validation can be performed by deploying the artifact in the field.

6 **Research Plan**

Various activities that are to be performed as part of the research process, along with time estimated to complete the same are listed in Table 1:
<table>
<thead>
<tr>
<th>SN</th>
<th>Research Activities</th>
<th>Estimated Duration (in person months)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Identification of CASE tool and Integrated Development Environments (IDE) that are customizable</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Customization of the CASE tools and IDEs</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>Identification of the features to be used in the calculation of the metrics that would help us calculate the evolution distance</td>
<td>2</td>
</tr>
<tr>
<td>4.</td>
<td>Design of the pattern base</td>
<td>1</td>
</tr>
<tr>
<td>5.</td>
<td>Identification of customizable requirements traceability tools</td>
<td>1</td>
</tr>
<tr>
<td>6.</td>
<td>Customization of the requirements traceability tools</td>
<td>3</td>
</tr>
<tr>
<td>7.</td>
<td>Identification of code pattern identification technique</td>
<td>1</td>
</tr>
<tr>
<td>8.</td>
<td>Adapting the code pattern identification technique to identify patterns in other artifacts</td>
<td>5</td>
</tr>
<tr>
<td>9.</td>
<td>Development of the Analyzer</td>
<td>5</td>
</tr>
<tr>
<td>10.</td>
<td>Development of the Abstracter</td>
<td>4</td>
</tr>
<tr>
<td>11.</td>
<td>Evaluation of MPRRS</td>
<td>7</td>
</tr>
<tr>
<td>12.</td>
<td>Preparation of an article for submission</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><strong>Total duration</strong></td>
<td><strong>34</strong></td>
</tr>
</tbody>
</table>

Table 1: Research Plan

7 Expected Contributions

The primary significant contribution of this research is the introduction of the notion of maintenance patterns. Even though Kemerer et al (Kemerer and Slaughter 1997) have discussed about their longitudinal study to find patterns of software evolution, they do not provide clear definitions as to what exactly they refer to as maintenance and evolution patterns. Since they allude to certain aspects of maintenance as patterns, viz., attributes of modules that are modified, their definition of maintenance patterns appears to be different from our approach. No prior research has dealt with automated identification, storage and retrieval of maintenance and evolution patterns, which is one other major significant contribution of this research.

This research will have a significant impact on practitioners by improving the process of maintaining legacy systems. Since patterns provide a way to reuse past experience, rectification of maintenance issues will become less expensive in terms of time and money. The fact that maintenance costs constitute a large proportion of software development budget, a reduction in this cost would lead to a considerable reduction in the total software development cost.

This would be a starting point for researchers to further investigate various means of pattern extraction so as to facilitate reuse. This would trigger further investigation into the usage of maintenance and evolution patterns during the development of new software systems. Since
these patterns provide a handle on the nature of problems that software maintainers face, this could be translated to generic guidelines to be followed during software development so as to alleviate the problems in the maintenance phase. Knowledge of evolution patterns that provide insights into the nature of software evolution can be used to develop systems with better interfaces so that they are adaptable for future evolution.

8 Research vs. Tool Development

Extension of knowledge that delineates an improvement over the status quo is considered to be research (March and Smith 1995). The artifact development proposed should not be seen as a mere tool development exercise, as the artifact embodies the prescriptions (March and Smith 1995) of this research and it is built to demonstrate feasibility of the method that we suggest that could improve the process of software maintenance. Since progress is achieved only when a technology is replaced by a more effective one (March and Smith 1995), we also suggest that the artifact be evaluated for effectiveness, based on valid metrics. This research extends knowledge in various aspects:

- Introduction of the notion of maintenance and evolution patterns
- Introduction of automated identification and retrieval of maintenance and evolution patterns
- Development of an artifact that embodies the prescriptions of this piece of design research

9 Limitations and Future Research

The capability of MPRRS is limited to the level of integration allowed by the interface to the legacy systems. Nevertheless, the users can intervene by providing inputs about their experiences formatted as maintenance patterns.

Future research could address the notion of anti-patterns with respect to software maintenance and evolution. Anti-patterns are ‘lessons learnt’ as opposed to patterns, which are ‘best practices’. This research can be extended by adding the capabilities to represent process patterns, which describe a collection of task/techniques/actions for successfully developing software (Ambler 1998), as applied to all phases of software development life cycle.
10 References


Gamma, E., R. Helm, et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.


**Appropriateness and Feasibility:** The topic chosen is appropriate and the research proposed is feasible, after some refinements, to be executed as a doctoral dissertation.

**Feasibility of this research as a doctoral dissertation**

1. **Economic feasibility:** Economic feasibility can be discussed by listing the resources required for executing this research and discussing whether it is accessible/affordable. Since past research has addressed the identification of code patterns, associated techniques could be adapted and used. Licenses are available for certain development tools (MS Visual Basic/Visual C++/Java) that might be required. CASE tool vendors provide affordable academic versions of their products, which can be used for our purposes. A relational database management system, required for developing the pattern base is available too. Requirements traceability tools could be available with the help of professors working in this area. Evaluation possibilities might be limited to the use of graduate students, as real-time deployment of the artifact developed might be difficult.

2. **Availability of Faculty members:** Effective execution of a doctoral dissertation would not be possible without the guidance of faculty members who are interested in the proposed area of research. I would request guidance from many such faculty members in the department who are working in the same or related area.

3. **Time constraints:** The objectives of the proposed research are ambitious and the research plan is very optimistic. A more thorough literature search is to be done to leverage the research already done, and the objectives should be refined according to the time available. The research plan indicates an approximate estimated effort of 34 person months, which is much higher than the time available for a typical doctoral student. The study could be split into two phases, system development and evaluation. The objectives should be refined so that these two phases can be traversed in multiple cycles.

**Appropriateness:** A study would be appropriate as a doctoral dissertation in IS if the following criteria are satisfied: It is suitable to the field of IS; It is not too trivial or not too large in scope; the topic is of interest to researchers and practitioners. This research is suitable to the field of IS, as it would have a great impact on the software maintenance and evolution process, which is an area of immense concern and interest to both practitioners and researchers. It falls under the IS Development domain as it involves the design of techniques, tools and methodologies for the efficient and effective development of IS.
Publication Strategy

I would focus on targeting three types of outlets:

(a) Conference proceedings

(b) A premier journal that has a wide practitioner audience

(c) A premier journal that has a wide research audience

After building a prototype, I would attempt to present the ideas in one or more conferences/workshops that focus on the area of software maintenance, in order to get feedback from the research community. IEEE International Conference on Software Maintenance\(^1\) is one such conference, which focuses specifically on the field of software and system evolution, maintenance and management. International Conference on Software Engineering\(^2\) would be a potential target too as software evolution, process improvement and reverse engineering are some of the areas that are covered by this conference. As this research focuses on automating some aspects of software maintenance and evolution, IEEE International Conference on Automated Software Engineering\(^3\) would be one other possible outlet. Pattern-oriented focus of this research makes it suitable to conferences focusing on development and application of patterns, viz., European Conference on Pattern Languages and Programs\(^4\). Since Software Engineering is one of the topics covered by the Workshop on Information Technologies and Systems (WITS), it would be one other potential target.

Premier journals that target practitioners and that are suitable for this research would be Communications of the ACM and IEEE Software. Researcher-Oriented premier outlets that I would target are IEEE Transactions on Software Engineering, Journal of Software Maintenance, and ACM Transactions on Software Engineering and Methodology.

\(^1\) [http://www.computer.org/cspress/CATALOG/pr00753.htm](http://www.computer.org/cspress/CATALOG/pr00753.htm)
\(^2\) [http://www.computer.org/cspress/CATALOG/pr08368.htm](http://www.computer.org/cspress/CATALOG/pr08368.htm)
\(^3\) [http://www.computer.org/cspress/CATALOG/pr00710.htm](http://www.computer.org/cspress/CATALOG/pr00710.htm)
\(^4\) [http://www.hillside.net/patterns/EuroPLoP/](http://www.hillside.net/patterns/EuroPLoP/)