Towards an Aspectual Analysis of Legacy Systems

Bedir Tekinerdo_an
Department of Computer Science
University of Twente
P.O. Box 217 7500 AE
Enschede, The Netherlands
bedir@cs.utwente.nl

Yasemin Satıro_lu
Department of Computer Science,
Bilkent University,
Bilkent 06800,
Ankara, Turkey
yasemins@cs.bilkent.edu.tr

Abstract

Aspect-Oriented software development provides explicit mechanisms for coping with concerns that crosscut many components and are tangled within individual components. Current AOSD approaches have primarily focused on coping with crosscutting concerns in software systems that are developed from scratch. In this paper we will investigate the applicability of AOSD to the evolution of legacy information systems. Various approaches have been already proposed to enhance LIS, however, these approaches have not explicitly considered crosscutting concerns and/or AOP techniques. We provide a categorization of legacy systems and give some early results in identifying and specifying aspects in legacy systems.

Keywords
Legacy information systems, aspect-oriented software development.

1. Introduction

A legacy software system may be defined informally as an old system that remains in operation within an organization [8]. Legacy systems typically have been developed several years ago sometimes without anticipating that they would be still running much later. Inevitably the software requirements for legacy systems might change and legacy systems must be evolved accordingly. Maintaining legacy systems, however, is in general hard because legacy systems very often run on obsolete, slow hardware that is hard to maintain, the documentation of the legacy system is lacking or incomplete, the interfaces of the legacy system components are limited for integration and/or adaptation, etc. Organizations dealing with legacy systems can either decide to replace the system or maintain the system. A simple replacement, if possible at all, might desirable but too expensive to consider because of the huge volumes of necessary changes, or too risky because of the continuous demand for on-line operation. For maintaining the legacy system techniques such as reengineering, reverse engineering, and software-reengineering patterns have been proposed [2].

These approaches have generally focused on, or are basically good at, coping with non-crosscutting concerns. Hereby, the maintenance and evolution requirements impact single components and can be more easily localized. In contrast, crosscutting evolution requirements have to deal with evolution of concerns that tend to crosscut several components. Required changes to these concerns are difficult because these changes need to be performed at multiple places impeding even further the maintainability. One basic reason why legacy systems are usually associated with high maintenance costs is because of the inflexibility of the adopted techniques [2]. In case, crosscutting concerns are not appropriately addressed, the continuous maintenance of legacy systems might thus easily lead to a degradation of its structure and as such it's maintainability.

The AOSD community has provided several general purpose solutions for coping with aspects in software systems [4]. Unfortunately, existing AOSD approaches seem to have primarily focused on identifying, specifying and implementing aspects for systems that are developed from scratch. Identifying, updating and specifying aspects in legacy information systems impose different requirements and constraints on the maintenance.

This paper aims to identify the applicability of AOSD for legacy systems. We think that this study is beneficial in two perspectives. First, AOSD researchers can get ideas for enhancing AOSD approaches to coping with legacy system code that is usually more difficult to access than systems developed from scratch. Second, maintainers of LIS can draw lessons from this study and use the techniques in this paper as complementary to the existing legacy code.

In particular we are interested in finding answers to the following questions:

1. *Which legacy systems could benefit from AOSD*

techniques?

2. *How to modularize crosscutting concerns in legacy systems?*

The paper is a first step towards finding an answer to the above questions. The remainder of this paper is organized as follows: Section 2 provides a categorization of legacy systems. Section 3 analyzes the existing legacy systems with respect to their ability to implement crosscutting. Section 4 presents an overall analysis approach for enhancing legacy systems with crosscutting concerns. Finally section 5 presents the conclusions.

2. Categorization of Legacy Systems

The term legacy system is overloaded and can be interpreted in different ways. To analyze the applicability of AOSD in improving the maintainability of legacy systems it is required that we understand which type of legacy systems exist. For this we have categorized legacy systems according to the criteria of *criticality to business needs*, *health state* and *accessibility*. These criteria have been derived from the literature on legacy systems.

2.1 Categorization based on Criticality

This categorization is done according to the business criticality criteria and regards legacy systems from an economical perspective. The legacy system types in this category are *mission critical* and *replaceable legacy system* types.

- *Mission critical* systems are the systems that are essential to the continued operation of the business, and, that provide service on which the organization is highly dependent. A failure in this type of systems may have a serious impact on the business [4]. If a mission critical system stops working, the business may grind to a halt. Also, these systems hold mission critical business knowledge which cannot be easily replaced [10].
- *Non-critical, replaceable systems* are the systems that no longer meet business needs or that are technically inefficient. These systems are ineffective in support of the business, and they are constantly falling over or becoming expensive to maintain.

2.2 Categorization based on Health State

This categorization is done according to the health state criteria. In this categorization, legacy systems are compared to living organisms. Their environment can affect their state of health; they can be more or less healthy depending on the changes in their environment and the treatment they receive from the organization they reside in. The legacy system types that are in this category are *healthy*, *ill*, and *terminally ill* legacy system types [9].

- *Healthy legacy systems* are the systems that satisfy the current enterprise needs and are kept healthy by routine maintenance. Routine maintenance is the incremental and iterative process in which small changes are made to the system. These small changes are usually bug corrections or small functional enhancements. Healthy systems don't need major structural changes in order to support business needs [5]. For these systems, either the legacy system is satisfactorily handling current enterprise needs or the needs are changing in relatively minor ways such that the legacy system can be updated or maintained in a timely and economical fashion [9].
- *Ill legacy systems* are the systems whose health has deteriorated to the point that some kind of non routine intervention is required [9]. For these systems, routine maintenance falls behind the business needs and a modernization effort is required. An ill legacy system requires more extensive changes than those possible during maintenance. These changes include system restructuring, important functional enhancements, or new software attributes [5].
- *Terminally ill legacy systems* are the systems that cannot keep pace with the business needs. The life of these systems can be prolonged by extraordinary life support, but heroic measures are required and are often not economically justified. That is, for these systems, modernization is either not possible or not cost effective, and these systems must be replaced. Also, if there is nobody left that knows anything about the system and there is no source code available for the system, the system is terminally ill [9].

2.3 Categorization based on Accessibility

This categorization is done according to the accessibility criteria. The legacy system types in this category are *black box* and *white box* legacy system types.

- *Black box legacy systems* are the systems that are like a black box; we have no detail on the internal structure of these systems. For these systems, only the externally visible behaviour is considered, not the implementation. Source code of the system is either not available or inscrutable. Also, a software component may be defined as a black box, if all the interactions occur through a published interface [4].

- *White box decomposable legacy systems* are the systems, for which the system internals, such as module interfaces, system components and their relationships, domain models are visible. The source code is available for these systems, and it is possible to extract information from the code in order to create abstractions that help in the understanding of the underlying system structure. In addition, the applications, interfaces, and database services can be considered as distinct components; and there are well defined interfaces for all these three components. In decomposable systems, there are no dependencies between the modules, such as procedure calls.
- *White box non-decomposable legacy systems*, are systems in which the system internals are visible but not separable. In essence, it is hard to derive the structure of the system.

3. Improving Legacy System Modularity using AOSD Approaches

3.1 Design Space of Legacy Systems

If we take the above categorization into account then we could classify each legacy system based on the given properties as illustrated in Table 1.

Table 1. Set of alternatives of mission critical legacy systems

	Health State	Accessibility	Maintenance Approach	Crosscutting implement.
1.	Healthy	Blackbox	Wrapping	DC: - SC: --
2.	Healthy	Whitebox-D	Wrapping	DC: ++ SC: ++
3.	Healthy	Whitebox-ND	Wrapping	DC: + SC: 0
4.	Ill	Blackbox	Migration, Wrapping	DC: -- SC: --
5.	Ill	Whitebox-D	Migration, Wrapping	DC: - SC: -
6.	Ill	Whitebox-ND	Migration, Wrapping	DC: - SC: -
7.	Terminal	Blackbox	Redevelopment	DC: -- SC: --
8.	Terminal	Whitebox-D	Redevelopment	DC: -- SC: --
9.	Terminal	Whitebox-ND	Redevelopment	DC: -- SC: --

This would result in a set of possible types of legacy systems, that is, the design space of legacy systems. A legacy system could be for example, mission critical because it is important from a business perspective, healthy since it can keep satisfying the business needs through conventional maintenance activities and white box decomposable because of a clear accessible structure. Given the three categorization dimensions we could have 2x3x3 = 18 possible kind of legacy systems.

In general legacy systems which are not business critical are usually not considered for maintenance activities. For this, we will consider only mission critical legacy systems which will lead to 9 possible legacy systems. These legacy system alternatives are listed in Table 1.

3.2 Conventional approaches

In principle, legacy systems are enhanced using one of the following techniques [3]:

- *Wrapping*, provides a new interface to a legacy component so that it can be more easily accessed by other components.
- *Migration*, moves the LIS to a more flexible environment, while retaining the system's original functionality.
- *Redevelopment* rewrites existing code. It requires the system to be shut down either during development or during replacement. Redevelopment can imply a total replacement but also *reengineering* is usually categorized as a *redevelopment* activity as most reengineering efforts propose complete system reimplementation [3].

Given a concrete LIS problem, it is not always possible to categorize the solution according to one problem [3] and often combinations of these techniques are used. Table 1 shows the possible maintenance approaches for each type of legacy system.

3.3 Applicability of AOSD

In principle AOSD approaches could be seen as part of each of the above three legacy maintenance techniques. Of course this categorization is too broad and does not help us in deciding how AOSD could help to improve the maintainability of legacy systems.

Our primary focus is on improving the maintenance of legacy systems for crosscutting concerns. Crosscutting in AOSD can be categorized as *Dynamic Crosscutting* and *Static Crosscutting,* which we will abbreviate as DC and SC respectively.

Static Crosscutting enables the developer to add fields and methods to existing classes, to extend an existing class with another. *Dynamic Crosscutting* enables the developer to define additional implementation to run at well defined points in the program.

Crosscutting in AOSD is implemented in *aspects,* which represents a modular unit of crosscutting concerns. Aspects define *pointcuts* and *advices*. A *pointcut* is a construct to capture *join points*. A join point is a well-defined point of execution in a program. An *advice* is

executable code for a pointcut.

We have performed an initial analysis of the legacy systems as presented in Table 1 and evaluated it with respect to:

(1) ability to implement dynamic crosscutting

(2) ability to implement static crosscutting.

In essence both types of crosscutting require some visibility of the legacy system. For dynamic crosscutting it is important to have some visibility to represent for example the pointcut specification. Without a proper view on the structure it is hard to identify the joinpoints and as such to specify the pointcuts. Dynamic crosscutting will be of course the easiest if the legacy system is redeveloped in which case the whole structure will be known again.

For static crosscutting the visibility of the structure of the system is even more important, especially when it is for example needed to extend the classes with new classes or to introduce new methods and fields. In that case it is important that the separate components of the systems can be viewed and accessed separately. This implies that we need preferable to deal with a legacy system that is whitebox and also decomposable.

Based on these informal guidelines we have assessed each legacy system type using the (increasing) scale --, -, 0, +, ++, with the meanings *very low, low, fair, high,* and *very high,* respectively. For example, in case the implementation of the crosscutting is not possible at all it was assigned a --. A ++ on the other hand means that the legacy system is very suitable for enhancing crosscutting concerns using AOSD techniques.

Although a quantitative analysis is very difficult in assessing the applicability of AOSD to legacy systems, we can still derive some useful heuristics that can be applied during maintenance activities. Looking at table 1 we could for example derive the following heuristics as presented in Table 2.

We could derive other rules from the given categorization. Of course these rules remain heuristics and can not always be very strictly applied. These are also mentioned as practical guidelines or rules of thumb to help the legacy maintainer in deciding whether AOSD can be applied to enhance the concerns.

***Table 2**. Heuristic rules for assessing the ability for implementing crosscutting in legacy systems*

1. IF health state is <healthy> AND
   accessibility is <blackbox>
   THEN
   ability for dynamic crosscutting is LOW AND
   ability for static crosscutting is VERY LOW.

2. IF health state is <healthy> AND
   accessibility is <whitebox-d>
   THEN
   ability for dynamic crosscutting is VERY HIGH AND
   ability for static crosscutting is VERY HIGH.

3. IF health state is <healthy> AND
   accessibility is <whitebox-nd>
   THEN
   ability for dynamic crosscutting is HIGH AND
   ability for static crosscutting is FAIR.

4. IF health state is <ill> AND
   accessibility is <blackbox>
   THEN
   ability for dynamic crosscutting is VERY LOW AND
   ability for static crosscutting is LOW.

5. IF health state is <ill> AND
   accessibility is <whitebox-d>
   THEN
   ability for dynamic crosscutting is MEDIUM AND
   ability for static crosscutting is LOW.

6. IF health state is <ill> AND
   accessibility is <whitebox-nd>
   THEN
   ability for dynamic crosscutting is VERY LOW AND
   ability for static crosscutting is VERY LOW.

7. IF health state is <terminal ill> AND
   accessibility is <blackbox>
   THEN
   ability for dynamic crosscutting is VERY LOW AND
   ability for static crosscutting is VERY LOW.

….

4. Approach for Analysis of Legacy Systems

The process for analyzing a legacy system is depicted in figure 1. First of all the legacy system will be analyzed based on its business criticality, health state and accessibility and based on this a characterization of the legacy system will be defined. In case we have to deal with a legacy system that is not maintainable, terminal ill, then the legacy system will either be redeveloped or dismissed. If the legacy system is still maintainable then the concerns that need to be enhanced need to be analyzed. Enhancement of a concern here means adding, updating or even eliminating concerns.

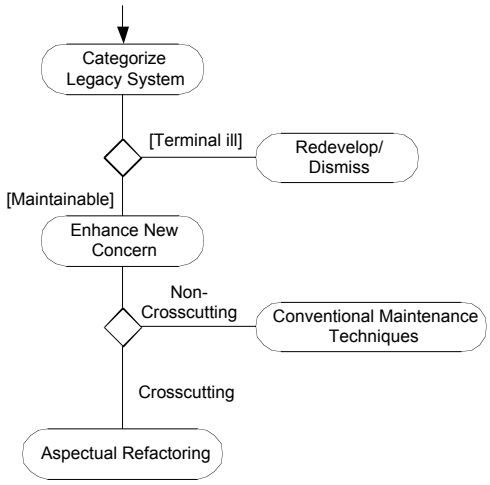

***Figure 1**. Approach for analyzing legacy systems for crosscutting concerns*

If the corresponding concern is a non-crosscutting concern then conventional legacy maintenance techniques can be used. If we are dealing with a crosscutting concern then, if possible, aspectual refactoring must be applied. Aspect-oriented refactoring has been studied by several other researchers [6]. Aspect-oriented refactoring can be applied to improve the understandability and the structure of either non-aspect code or existing aspect-oriented code. In our study we are interested in the first one, and in particular the refactoring of object-oriented legacy code. We could for example apply migration techniques for an object-oriented legacy system and refactor this to an aspect-oriented version of the system. Several aspectual refactoring patterns have been identified such as *extract advice, extract implementation, extract interface implementation, extract exception handling* etc.

5. Conclusion

This paper provides a first initial attempt for analyzing the applicability of AOSD to legacy systems. For this, we have provided a categorization of existing legacy systems and analyzed these with respect to the ability for implementing static and dynamic crosscutting. Although the results are still immature they provide us the vision for our future research activities. Our future work will enhance the heuristic rules for analyzing legacy systems and include the development of a tool for guiding the legacy maintainer in applying AOSD.

References

[1] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London: Academic Press., 1985.

[2] K. Bennet. "Legacy System: Coping with Success". IEEE Software, pp. 19-23, January 1995.

[3] J. Bisbal, D. Lawless, B. Wu and J. Gromson. *Legacy Information Systems: Issues and Directions*, IEEE Software, Vol. 16 No. 5 pp.103-111, September/October 1993.

[4] T. Elrad, R. Fillman, & A. Bader. *Aspect-Oriented Programming*. Communication of the ACM, Vol. 44, No. 10, October 2001.

[5] J. Hannemann and G. Kiczales. *Overcoming the Prevalent Decomposition of Legacy Code*. *In Workshop on Advanced Separation of Concerns*, 2001.

[6] S. Hanenberg, C. Oberschulte, R. Unland. Refactoring of Aspect-Oriented Software. NetObject Days. 2003.

[7] R. Laddad. Aspect-oriented Refactoring Series. TSS. 2003.

[8] I. Warren. *The Renaissance of Legacy Systems*. Practitioner Series. Springer Verlag, 2000.