Contents

Foreword to the Poster Session By the General Chair

We are very proud to be able to include an excellent poster session in this year's International Conference for Software Maintenance. Poster sessions serve an important function in conferences of this type since they give the attendants an opportunity to discuss with the poster presenters. Posters are intended to provoke a discussion and to help conference participants learn from one another. The formal presentations are one means of communication, the informal discussions provoked by the poster sessions are another.

The hosting hotel of the conference – the Budapest Thermal Hotel on Margaret's Island – with it's many lobbies, health facilities and open cafes is particularly suitable for such small group discussions. We are integrating the poster presentations with the coffee breaks so that all participants will have the opportunity to take part in the discussions. We trust that these discussions will be fruitful for both parties – the presenters and the attendants.

In any case, the array of porters promises to be very informative. They include such hot topics as aspect-oriented programming, predicting software evolution trends, software measurement, maintenance processes, dealing with duplicate code, program comprehension, reengineering software architecture, assessing maintenance risks and extending dynamic linking for program customization. I heartily recommend everyone taking part in these vital discussions.

Finally, I want to thank the authors of the poster presentations for their contribution to the technical excellence of this year's conference and to their help in making this a milestone event in promoting the significance of software maintenance and evolution.

Harry Sneed
General Chair

Program overview

Saturday (Sept 24)	**STEP**		
Sunday (Sept 25)	**STEP, VISSOFT**		
Monday (Sept 26)	**WSE, Software Evolvability, Industrial Truck, Tutorial**		
Tuesday (Sept 27)	**ICSM** 8:30am-11:00am Registration 9:00am-10:30am Opening		
	ICSM- Technical sessions 11:00am-6:00pm T1: Aspect Mining T4: Maintenance T7: Maintenance in Practice	**ICSM**- Technical sessions 11:00am-6:00pm T2: Components & Frameworks T5: Re- and Reverse Engineering T8: Process	**ICSM**- Technical sessions 11:00am-6:00pm T3: Distributed Systems T6: Source Code Analysis T9: Program Comprehension
	ICSM 7:00pm Banquet Dinner in the Grand Hotel Restaurant		
Wednesday (Sept 28)	**ICSM** 9:00am-10:00am Keynote		
	ICSM- Technical sessions 9:00am-6:00pm T10: Feature Extraction and Analysis T13: Theoretical Maintenance T15: Evolution	**ICSM**- Technical sessions 9:00am-5:30pm T11: Refactoring T14: Testing I. T16: Testing II.	**ICSM**- Technical & Panel 9:00am-5:30pm T12: Regression Testing P1: Panel 1 P2: Panel 2
	ICSM 7:00pm Csárdás Evening in the Wine Cellars of Budafok		
Thursday (Sept 29)	**ICSM** 9:00am-10:00am Keynote		
	ICSM- Technical & Short papers 9:00am-3:30pm T17: Web maintenance & Reengineering S3: AOP & Web	**ICSM**- Short papers 9:00am-3:30pm S1: Maintenance & Evolution S4: Testing	**ICSM**- Short papers & Dissertation 9:00am-3:00pm S2: Program Comprehension D1: PhD Dissertation session
	ICSM 3:30pm Closing		
Friday (Sept 30)	**ICSM** 8:30am-4:00pm Bugac puszta excursion	**SCAM**	
Saturday (Oct 1)	**SCAM**		

Architectural level Maintainability Based Risk Assessment[1]

W.Abdelmoez, I. Shaik, R. Gunnalan,
M. Shereshevsky, K. Goseva-Popstojanova, H.H. Ammar
Lane Department of Computer Science,
West Virginia University
Morgantown WV 26506
{rabie, isrars, gunnalan, smark, katerina, ammar}@csee.wvu.edu

A. Mili
College of Computer Science,
New Jersey Institute of Technology
Newark NJ 07102
mili@cis.njit.edu

C. Fuhrman
Department of Sofware and IT Engineering
École de technologie supérieure
Montreal, Canada H3C 1K3
christopher.fuhrman@etsmtl.ca

Abstract

Software maintenance accounts for a large part of the software life cycle cost. Systems with good maintainability can be easily modified to fix faults or to adapt to changing environments. In this paper, we define maintainability-based risk as a combination of two factors: the probability of performing maintenance tasks and the impact of performing these tasks. An estimation procedure based on change propagation probabilities from architectural artifacts is presented. We illustrate how to apply the procedure on a case study which uses design patterns to improve system quality. This type of risk assessment helps in managing software maintenance process. It can be used to identify the most risky parts of the system and assign them to the most experienced maintainers.

Keywords: Maintainability-based risk, software architectures, software metrics, software maintenance, change propagation probability.

1. Introduction

Software maintenance is defined as *modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment* [8]. Basically, there are three types of software maintenance; corrective maintenance deals with error corrections, perfective maintenance tries to improve the quality of the system, and adaptive maintenance concerned with system changes as requirements and environment changes. Software maintainers usually are not involved in the original software development cycle. They must learn how a program functions before they can change it. They often interact with complex and difficult to comprehend systems. The status of system documentation, programmer skill and experience and the attributes of the system itself are some of the variables that affect the maintenance process. According to [14], the cost of software maintenance averages from 60% to 80% of the overall software system cost. Furthermore, enhancements (perfective and adaptive maintenance) account for 78%-83% of the maintenance effort. As a result, maintainability is an important software quality factor. IEEE computer society defines maintainability as *the ease with which a software system or a component can be modified to correct fault, improve performance or other attributes, or adapt to a changed environment* [9].

The research effort aimed at quantifying software maintainability is rather limited. Moreover, to the best of our knowledge there are no attempts to quantify maintainability-based risk for software systems. In general, risk assessment provides useful means for identifying potentially troublesome software components that require careful development and allocation of more testing effort. According to NASA-STD-8719.13A standard [12] risk is a function of the anticipated frequency of occurrence of an undesired event, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity. This standard defines several types of risk, such as for example reliability risk, availability risk, acceptance risk, performance risk, cost risk, schedule risk, etc.

In this paper, we focus on perfective maintenance and refactoring activities in particular. Refactoring is defined as *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviors.* Refactoring to improve the design of the system requires knowing which parts of the system need to be improved. In [5], Fowler and Beck presented a list of bad smells that

[1] This work is supported by the National Science Foundation through ITR program grant No CCR 0296082, and by NASA through a grant from the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA Independent Verification and Validation (IV&V) Facility, Fairmont, West Virginia.

help to identify where refactoring is needed. Examples of bad smells include large class, lazy class, data class, and switch statements. It should be noted that not all of the smells can be identified at the architectural level.

One way to refactor the software is to use design patterns. According to [10], there are practical situations where patterns help to improve the quality of the design. The intent is to improve maintainability by reducing or removing duplication, simplifying what is complicated and making the design development better at communicating. Design pattern is a software engineering problem-solving discipline. It has roots in many disciplines, including literate programming, and most notably in Alexander's work on urban planning and building architecture [3]. The goal of the pattern community is to build a body of literature to support design and development. The Gang of Four's book [6] presented the first well-described and documented catalog of design patterns for object-oriented design.

In our research effort, we are concerned with maintainability-based risk that takes into account the probability that the software product will need to endure a certain type of maintenance and the consequences of performing this maintenance on the system. Maintainability-based risk can be used to improve the maintainability of the system architecture, to manage system maintenance process, and to identify the risky components of the system in terms of maintainability. This study is part of a wider effort that considers other types of architectural level risk such as reliability-based risk [7] and performance-based risk [4].

The rest of the paper is organized as follows. In Section 2, we briefly discuss the literature background of our study. In Section 3, we define how maintainability-based risk can be estimated based on change propagation probabilities and illustrate our estimation procedure on a case study using design patterns as means of refactoring. Finally, in Section 4, we summarize our results and discuss the directions for future research.

2. Background

2.1. Software Maintenance Risks

Many types of risk are ushered when software systems undergo maintenance. They are similar to those we face when developing new software systems, but with different level of risk. These types of risk are [16]:

- *Project risk*— Maintenance project cannot be carried out within the budget or on time.
- *Usability risk*— Systems will cause problems and failures after the maintenance is conducted.
- *Maintainability risk*—It will be difficult to maintain the system in the future because of the way we conducted this maintenance.

In this paper, we focus on maintainability risk. In particular perfective maintenance achieved by refactoring activities.

There have been some studies trying to characterize and quantify software maintainability. One of the famous studies [13] introduced the Maintainability Index (MI) measure which is calculated using a polynomial of widely used code level measures such as Halstead measures and McCabe's cyclomatic complexity. In [11], Muthanna et al. conducted a similar study which used design level metrics to statistically estimate the maintainability of software systems. They constructed a linear model based on a minimal set of design level software metrics to predict Software Maintainability Index. In [15], Prechelt et al. conducted a controlled experiment to study the effect of applying design patterns on maintenance effort. They concluded that unless there is a clear reason to prefer a simpler solution, it is probably wise to choose the flexibility provided by the design pattern.

2.2. Change Propagation

The estimation procedure of maintainability-based risk builds on our previous work on change propagation probabilities [1]. Let us consider a software architecture modeled by components and connectors. We are interested in the maintainability of the products instantiated from it. In corrective or perfective maintenance tasks, *change propagation probability* matrix for an architecture reflects on the probability of changing component C_j as a result of a change to component C_i. The estimation of the elements cp_{ij} of the change propagation matrix CP is based on the following definition [1] :

Definition. Given components C_i and C_j of a system S, the *change propagation probability* from C_i to C_j is denoted by cp_{ij} and defined as the following conditional probability

$$cp_{ij} = \mathbf{Pr}(([C_j]\neq[C_j'])|([C_i]\neq[C_i'])\wedge([S]=[S'])), \quad (1)$$

where [X] denotes the functionality of component/system X and S' is the system obtained from S by changing C_i into C_i' (and possibly C_j into C_j' as a consequence).

To estimate the change propagation probabilities, we first analyze the architecture of the system under investigation using a structural diagram or a class diagram. From these artifacts, we can identify the components and the connectors of the component-based system architecture. Then, we analyze message protocols between every pair of components in the system which provide us with the messages exchanged between components C_i and C_j. With the help of case tools, we can get message sets for any pair of components in the system [2]. This information can also be obtained from static analysis tools of the source code.

An architecture can be seen as a collection of components C_i, i=1,…,N. With every component C_i, we associate the set V_i of the interface elements of the provided functions of C_i. We determine the *usage coefficient* value π_v^{ij} for every interface element $v \in V_i$ and every other component C_j, $j \neq i$. They take binary values:

- π_v^{ij} =1, if the interface element ν provided by C_i is required by C_j. This means that any signature change in component C_i associated with interface element ν will propagate to component C_j.
- π_v^{ij} =0, otherwise.

Hence, for every pair of components C_i and C_j, i≠j, the change propagation probability cp_{ij} can be estimated based on the values of the *usage coefficients* π_{ivj} by [1] :

$$cp_{ij} = \frac{1}{|V_i|} \sum_{v \in V_i} \pi_v^{ij} , \qquad (2)$$

3. Maintainability Based Risk

We define maintainability-based risk as a combination of two factors: the probability of performing maintenance tasks and the impact of performing these tasks. In this paper, we limit our scope to refactoring activities that are used to reorganize the system in order to make it more adaptable to add requirements or improve its quality. Accordingly, maintainability-based risk for a component is defined as:

*Probability of changing the component * Maintenance impact of changing the component.*

We use bad smells of the architecture to estimate components' maintainability-based risk. In particular, we consider two smells: divergent change and shotgun surgery.

Divergent change is when one component is commonly changed in different ways for different reasons [5]. For example, we have to modify the same component whenever we change the database or add a new calculation formula. To identify a divergent change smell using the CP matrix, we examine if we have high values in a column corresponding to a component C_i. Such a component is likely to undergo frequent changes in the maintenance phase due to changes in other components.

Shotgun surgery is a phenomenon when every time a change is made to a component many little changes need to be made to a lot of different components [5]. For example, whenever we change a database we must change several components. To identify a shotgun smell using the CP matrix, we examine if we have high values in a row corresponding to a component C_i. Changes to such a component need to be avoided because they propagate throughout the system.

Thus, maintainability-based risk is proportional to:

α Divergent change * Shotgun surgery

α Average (columns CP) * Sum (rows CP)

$$\alpha \ \frac{1}{N} \sum_{i=1}^{N} cp_{ij} \ . \sum_{j=1, j \neq i}^{N} cp_{ij} \qquad (3)$$

We use equation (3) to estimate components' maintainability risk of the original system and the refactored system after applying a design pattern. As a case study, we use an open source calendar and task tracking software written in Java [17]. The design of the calendar depends on MVC (Model view controller) design pattern. We studied two versions of the calendar. The first version implements only the view and the model of the MVC design pattern. The second version incorporates the controller. Due to the space limitation, only the results of the estimation procedure are presented. The details of the case study will be presented in the poster.

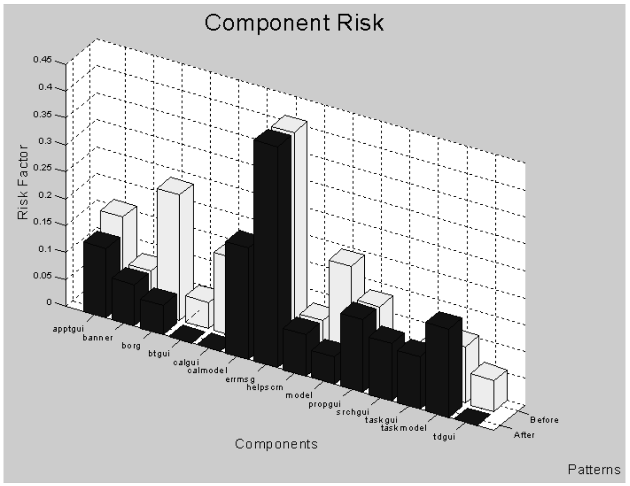


Figure 1. Components maintainability-based risk for the case study

Figure 1 shows components maintainability-based risk for the case study before and after implementing the controller of the MVC pattern. We can identify that *errmsg* is the most risky component. This component is responsible of showing an error message whenever an exception occurs. The risk factor of this component is not affected by adding the controller class of the MVC pattern because this modification does not address the *errmsg* component. However, there are improvements in the maintainability-based risk of some components (e.g. *borg, apptgui, and tdgui)*. We restrict the analysis to the components that existed before adding the controller to the MVC pattern. The biggest improvement in maintainability risk factor is in *borg* component. This is a result of adding the controller class, as it causes the coupling of the borg component to decrease because it is redirected to the added controller class. On the other hand, this modification causes other components to be more coupled (e.g. *taskmodel* and *taskgui*) because they need to interact with the added controller class. As a result the maintainability risk factor of these components is increased.

4. Conclusion

In this paper, we introduce the concept of architectural level maintainability-based risk assessment. Maintainability-based risk is defined as a combination of two factors: the probability of performing maintenance tasks and the impact of performing these tasks. We present an estimation procedure based on change propagation probabilities using architectural information of the system. We discuss its capability to assess the effect of applying design patterns on the components maintainability. Maintainability-based risk assessment can be used to guide software maintenance management. Also, it can identify the risky parts of the system, so that they can be assigned to the most experienced maintainers.

Among our venues of future research, we plan to carry out more experiments to examine how other design patterns affect the maintainability-based risk of the software components. We also plan to automate the computation of the maintainability-based risk by extending the Software Architectures Change Propagation Tool (SACPT) [2].

References

[1] AbdelMoez W., M. Shereshevsky, R. Gunnalan, H.H. Ammar, Bo Yu, S. Bogazzi, M. Korkmaz, A. Mili , "Quantifying Software Architectures: An Analysis of Change Propagation Probabilties", ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 05), Cairo, Egypt, January 3-6, 2005.

[2] Abdelmoez W., R. Gunnalan, M. Shereshevsky, H.H. Ammar, Bo Yu, M. Korkmaz, A. Mili, "Software Architectures Change Propagation Tool (SACPT)", Proc. 20th IEEE International Conference on Software Maintenance (ICSM 2004), Chicago, IL, September 2004.

[3] Alexander, C., S. Inshikawa, M. Silverstiein, M. Jacobson, I. Fiksdahl-king, and S. Angel. "A Pattern Language", Oxford University Press, New York, 1977.

[4] Cortellessa V., K. Goseva-Popstojanova, K. Appukkutty, A. Guedem, A. Hassan, R. Elnaggar, W. Abdelmoez, and H. Ammar, "Model-Based Performance Risk Analysis", *IEEE Transaction on Software Engineering,* Vol.31, No.1, January 2005, pp.3-20.

[5] Fowler M.and Beck K, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2004.

[6] Gamma E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Object-Oriented Software", Addison-Wesley, 1995.

[7] Goseva-Popstojanova K., A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, A. Mili, "Architectural-Level Risk Analysis using UML", IEEE transaction on software engineering, Vol.29, No.10, October 2003, pp. 946-960.

[8] *IEEE Standard for Software Maintenance*, The Institute of Electrical and Electronics Engineers, Inc.,New York, 1998.

[9] *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, Inc.,New York, 1990.

[10] Kerievsky J., Refactoring to Patterns, Addison-Wesley, 2004.

[11] Muthanna S., K. Ponnambalam, K. Kontogiannis and B. Stacey, "A Maintainability Model for Industrial Software Systems Using Design Level Metrics", Seventh Working Conference on Reverse Engineering (WCRE'00), Brisbane, Australia, November 23 - 25, 2000

[12] NASA Technical Std. NASA-STD-8719.13A, *Software Safety*, 1997. http://satc.gsfc.nasa.gov/assure/nss8719_13.html

[13] Oman, P. & Hagemeister, J. "Constructing and Testing of Polynomials Predicting Software Maintainability." Journal of Systems and Software 24, 3 (March 1994): pp. 251-266.

[14] Pigoski T.M., *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley & sons, 1996.

[15] Prechelt, L.; Unger, B.; Tichy, W.F.; Brossler, P.; Votta, L.G.; "A controlled experiment in maintenance: comparing design patterns to simpler solutions", IEEE Transactions on Software Engineering,Vol. 27, No. 12, Dec. 2001, pp.1134-1144

[16] Sherer S., " Using Risk Analysis to Manage Software Maintenance ," Software Maintenance: Research and Practice, Vol. 9, 345-364, 1997.

[17] Source Forge Project: BORG Calendar http://sourceforge.net/projects/borg-calendar/

Visualizing Dynamic Data Dependences as a Help to Maintain Programs

Françoise Balmas Harald Wertz Rim Chaabane
Laboratoire Intelligence Artificielle
Université Paris 8
93526 Saint-Denis (France)
{fb,hw,lysop}@ai.univ-paris8.fr

Abstract

This paper is on a project to evaluate the impact of visualizing dynamic data dependences in the context of maintenance activities.

Our work is based on previous work in displaying static data dependences and on experience with large sets of dependence displaying strategies that we adapted to deal with problems where dynamic information is crucial. We developed a prototype around a Lisp interpreter and applied it to a highly complex AI program. This permitted us to build efficient visualizations and to evaluate the benefits of using dynamic dependences for program understanding, debugging and correctness checking.

In this paper, we present our prototype, detailing especially the different visualizations we introduced to allow users to deal with hard to understand programs, and we discuss how dynamic dependences permit to see *what really happens during program executions.*

1. Introduction

In the past, we used static data dependences to help understand and document programs [1], and developed displaying strategies to deal with large sets of dependences [2]. In this context, we showed that visualizing sample values of variables, for a well chosen execution, was very efficient to help understand what a program does and how it works. That's why we decided to explore computing dynamic dependences and to evaluate the benefits of visualizing them for those activities where knowledge about given executions is crucial, that is program understanding, debugging and correctness checking.

For the sake of evaluation, we developed a prototype around the Lisp language; actually, modifying an interpreter is much easier than modifying a compiler, and hard to understand Lisp programs are still small enough to prevent algorithmic and optimization problems which arise when manipulating huge amounts of data.

To evaluate our approach, we applied our tool to a version of the classical AI Blocks World program [4]. In our version, the world is a table with different objects on it which can be manipulated by a one-handed robot. Basically, the program presents itself as an interpreter the user interacts with in order to create objects, make the robot move them to other places or ask for information about the current state of the world.

The program is around 1200 LOC long[1] and includes more than 125 functions and macros, many global variables modified through pointers, indirect recursive calls, thus long circularities, and escapes (i.e. non standard return controls). It evolved over time, since first developed for teaching purpose and then modified several times to add further reasonning capabilities. All these features make this program rather complex, hard to understand for newcomers to the program and difficult to maintain for the one of us who developed it.

In this paper, we report on this evaluation, discussing both the different kinds of visualizations we defined and the way they let us *see* what happened during execution of our program, helping us to understand, debug and check it for correctness.

2. The tool

Our tool relies on three modules: a modified Lisp interpreter (a C version is under construction), a database (currently a Lisp program) and a GUI (implemented in Tcl/Tk). We modified a Lisp interpreter to make it, in addition to normal execution of programs, extract dependences at runtime. These dependences are sent to a Lisp program that acts as a database, storing the dependences and producing,

[1]Note that LOC in Lisp is very different from LOC in more usual programming languages such as C, because of the compactness of code and the powerfull functional primitives it offers.

```
(de square (a)
   (* a a))

(de som2 (x y)
   (+ (square x) (square y)))

? (som2 3 5)
= 34
```

Figure 1. Sample code

on demand, the corresponding graph – in *dot* [3] format. Finally, a Tcl/Tk GUI displays the graph, using mechanisms to reduce its size, and allows users to interact with it to tune several kinds of visualizations.

The full set of dependences for a given call is unlikely to be displayed as is, since it is usually to large to be readable. We thus *group* together nodes (that is pieces of code) belonging to the same function call. For example, in the sample code of Fig. 1, wich computes the sum of the square of two numbers, we have nodes belonging to the two calls to function square and we aggregate them to form two groups. These two groups, as well as other nodes, belong to function som2 and are aggregated to form the main group. We can then display dependences showing only these groups, thus only the calls, and the dependences between them. Fig. 2 gives the corresponding graph for the call (som2 3 5) and shows how values are transmitted between calls. Alternatively, we can also get a graph with only the toplevel call visible (see Fig. 3), showing just input and output of the whole program. Such views are very helpful when global variables are used and modified by the program (see Section 4).

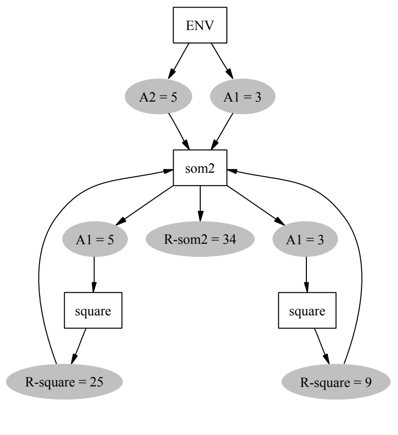


Figure 2. Data dependence graph with all calls visible

For a large program, the number of function calls may become also too large to get readable graphs. For this reason, we introduced a tagging mechanism to classify functions into control structures (they are functions in Lisp), primitives (those standard functions that are implemented in Lisp itself) routines (small reusable functions related to the program at hand) and user functions (all the remaining functions). The next Section will show different visualizations that depend on this classification to filter out given set of calls.

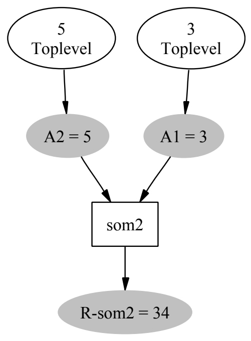


Figure 3. Data dependence graph with only the toplevel call visible

3. Visualizations

In this section, we introduce the different visualizations our tool offers to help users analyze programs. The first four are variants of call graphs: we noticed that navigation inside data dependences graphs is often tedious and call graphs provide a good 'map' to support this navigation. The last four visualizations are variants of data dependence graphs.

Call graph Such a visualization offers a global overview of the functions the program evaluted and the way they are organized. It also permits the user to ask for a given data dependence graph by interactively selecting a call: this call then becomes the *focus* of the displayed data dependence graph (see below).

User call graph This is a restricted version of the call graph just described, where only user functions are shown. This permits to get a graph with much fewer calls – from more than 3600 calls in the whole call graph for a 'move-object' instruction to our robot we could get down to about 30 calls –, thus more easily readable. This also permits to get a global overview of the main function calls from a programmer's conceptual perspective.

One level user call graph This visualization is a mix of the two previous ones: a call graph beginning at a given user function and ending at the next call of a user function. That is: when traversing the call tree, we stop drawing the graph when we reach leaves or we encounter user functions. This visualization gives all necessary details but locally bounded by user functions.

Return graph The Blocks World program uses intensively the 'escape' mechanism of Lisp[2] that allows the program control to directly return to a calling function up in the call tree. It is then often hard to conceptually follow where the control is supposed to get back and how the program is supposed to continue after the activation of the 'escape'. That's why we integrated the possibility to extend the *call* graphs with the *return* graph: whenever control gets back to another function than the one that called the current one, the return arrow is displayed in red.

Data dependence graph This visualization provides the standard data dependence graph we introduced in Section 2, with either only the toplevel call, or all calls. It may focus on a given call, this way considering only the sub tree beginning at this call.

Filtered data dependence graphs This visualization is obtained whenever classes of functions are tagged to be filtered out. It is especially useful with data dependence graphs where all calls are to be displayed, since it permits to hide functions of lesser interest for the task at hand. For example, to focus on the dependences from a programmer's conceptual perspective, it is useful to filter out control strucutres, primitives and routines that often fill a graph with irrelevant information.

First level graphs The two basic possibilities to examine calls – only the toplevel call visible, or every call visible – proved to be insufficient in several cases, since giving either too few or too many details. We extended our tool functionnalities with a view where the function call focused upon is visible along with each first level call. This allows the user to examine how a given action – implemented by a function call – is decomposed into smaller actions, without the need to examine the actual code of the call.

Sets of calls Sometimes, the automatically built views we just described are not satisfying because centered on *one* call, while we might need the ability to see a *set* of specific calls, especially to examine the values of global variables before and after these different calls (see Section 4). For this reason, selecting a few calls on a call graph results in a data dependence view where only these calls are shown while all others are hidden.

[2]Sometimes called 'catch-and-throw', this mechanism is similar to the 'setjmp-longjmp' mechanism of C.

The different visualizations presented in this section were inspired by the needs we encountered during the process of trying to understand and evolve a rather large and complex program. They showed to be very useful for interactive goal-directed exploration. In the next Section we will discuss more specifically the use of dynamic data dependences for different programming activities.

4. Dynamic data dependences for programming activities

Program discovery The first context where our visualizations proved to be useful is program discovery, that is the task a programmer faces when s/he has to get aquainted with a program s/he didn't implement her/himself. Even if interacting with the robot, on the Lisp terminal, was easy to grasp, trying to understand how the program works in order to handle object creation, placement and moving was another question!

A data dependence graph focused on the toplevel call is a good view to start with, since it shows how global variables are modified during the call. For example, Fig. 4 shows how the table, the object list and the object itself are modified during the creation of an object, showing this way the real effect of the call. The user-call-graph permitted us then to get a global overview of the actions performed, while tuning data dependence graphs for these different actions gave us futher information on how they affect the global variables.

Finding bugs While working on the discovery of the Blocks World program, we encountered graphs with wrong variable values, incorrect number of function calls or unexpected calls. Refining different graphs, we could navigate backward and forward to find the source of the problem.

For example, we noticed that when finding a place where to put a square object of 2x2, the robot only checked three positions on the table, while of course it should have checked at least four. This example shows that our visualizations may direct users to problems they don't even suspect: we didn't *search for* any problem in finding a place, we just *saw* there was one. We could then detect more precisely why there was this problem and how to solve it.

Correctness checking As an extension of the two former points, we also used our views to verify that the program was behaving properly. For instance, careful inspection of visualizations of the program's execution after correction of the 'finding place' bug permitted us to *see* its correctness.

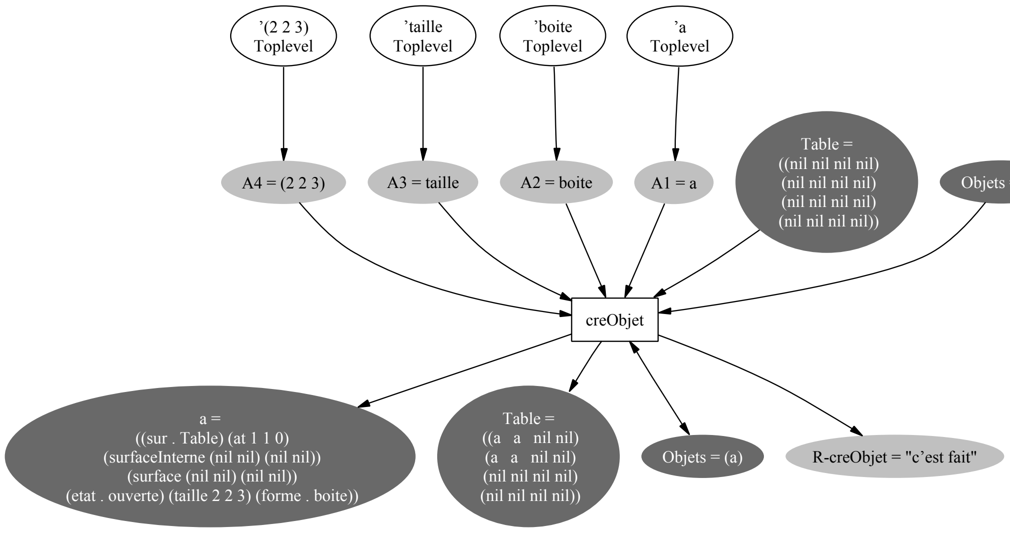


Figure 4. Overview of computation performed

We also used our views to verify that the program was behaving the way we expected it to do. As usually in AI programs, in many contexts large parts of the program – a function along with every call it performs – are reused and reused again, resulting in deep and broad call trees, extremely difficult to capture. To check that such functions were correctly implemented, we looked at user call graphs to check whether they were recursively called the correct number of times. We also looked at data dependence graphs where we rendered visible only calls to these functions, to examine the values of the global variables at the different steps of the program execution and to verify that they were modified the way we expected.

5. Conclusion

From our experience working with the Blocks World program, as well as several other programs, we can affirm that the major benefit given by the dynamic dependences our tool handles is that precise information about a program execution is recorded and visible, after execution, for examination: details about how execution was driven from one expression to another, as well as about the values variables had at any point of the program and how these values are transmitted from point to point.

The different visualizations we propose were designed to minimize the conceptual overload in order to allow users to *see* the exact information they need, otherwise barely accessible in the database. Clearly, dynamic information is of great help when working on problems like debugging, verifying that a program works properly, or even optimizing, since it gives information only for *one* given execution, when static dependences would give too much information.

On the other hand, the weakness of this approach is that it requires enough knowledge from the user on the possible paths in the programs: verifying that a program behaves properly means checking *many* possible executions, and the user has to find which ones are necessary. However, our approach also makes possible to discover some unforseen execution pathes, as static analysis would. Combining static information with dynamic dependences is one of our main perspectives. Two other perspectives are to extend our filtering mechanism to global variables – when they are not of interested at the same time, filtering out some would be useful – and to develop a query language permitting us to find, thus to jump to, parts of the execution corresponding to given criteria.

References

[1] F. Balmas. Using dependence graphs as a support to document programs. In *Proceedings of the Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, 2002.

[2] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal on Software Maintenance and Evolution: Research and Practice*, 16(3):151 – 185, May/June 2004.

[3] E. Koutsofios and S. North. *Drawing graphs with* dot. AT&T Labs – Research, Murray Hill, NJ, March 1999.

[4] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.

Aspect-Oriented Programming based Software Evolution with Microsoft .NET

Feng Chen and Hongji Yang
Software Technology Research Laboratory
De Montfort University, Leicester, UK
fengchen, hyang@dmu.ac.uk

He Guo and Tianyang Liu
Computer ScienceDepartment
Dalian University of Technology, Dalian, China
guohe@dlut.edu.cn

Abstract
Software system evolution often requires adding new general functions, which are distributed in many components of the system. A normal method is to insert code into every corresponding class, which is a trivial task. It may also increase the risk of introducing errors and destroy the structure of the system. In this paper, an Aspect-Oriented Programming (AOP) based software evolution approach with Microsoft .NET is introduced. By utilising 'Joinpoints', the proposed approach can insert new code into the evolving system without any modifications to the existed class structures. A prototype tool is developed for supporting the system evolution and case studies are used for illustrating and testing the proposed approach. Finally, a conclusion is drawn, which shows that the proposed approach is feasible and promising in its domain.
Keywords: *Aspect-Oriented Programming, Software Evolution, Microsoft .NET, Dynamic Weaving.*

1. Introduction

A software system is often required to add new general functions, which are distributed into many components of the system, at the stage of maintenance. There are many disadvantages if these functions are inserted into every needed place directly: firstly, it is too complex to do so; secondly, it may dramatically increase the risk of introducing errors into the software system. If k out of m modules are modified, the number of module interface checks required, N, is $N = (k*(m-k) + k*(k-1))/2$, which means that more testing needs to be done [11]. Furthermore, such an approach may destroy the structure and the encapsulation of the system, which will lead to 'tangled' code. In this paper, an AOP-based software evolution approach with Microsoft .NET is introduced. By deploying 'Joinpoints', the proposed approach can insert new code into the evolving system without any modifications to the existed class structures.

The remainder of this paper is organised as follows: Section 2 reviews characteristics of software evolution and AOP. Section 3 proposes an approach to AOP based software evolution with Microsoft .NET. In Section 4, prototype tool is demonstrated. In Section 5, case studies are used for illustrating and testing the approach. Finally, conclusion is drawn and further research directions are discussed.

2. Software Evolution and AOP

2.1 Characteristics of Software Evolution

Software evolution is defined as a kind of software maintenance that takes place only when the initial development was successful [1]. Much attention should be paid to the following principles when a software system is evolved:

- the evolved system should be reliable,
- the evolved system should be functional,
- the evolved system should be efficient, and
- the cost of the evolution should be acceptable.

2.2 AOP

In 1997, a new programming methodology, Aspect-Oriented Programming [8], was proposed. The core concept in AOP is the Joinpoint, which is first mentioned in AspectJ and is a well-defined point in the execution of a program-like method calls, loop beginnings and object constructions [13]. AOP enables a programmer to modularise common behaviours and encapsulate them in a new component [4, 8], which can be coded and revised independently and be injected into the existing component code with a 'weaver' [7]. This kind of injection can be either static or dynamic.

2.3 AOP Based Software Evolution

When a proposed evolution requires changes to more than one module, it is said to be crosscutting evolution. The need to address crosscutting evolution is crucial in software product lines as a change can affect different variants and branches [10]. The tasks of software evolution involve both the analysis of the source code and the injection of the new functions. Separating the aspects of systems that perform different roles may have many benefits for software development [14].

AOP supports evolution via crosscuts, which are sets of events (method calls, exception raises, etc.) that are to be intercepted, and Advice that is to be executed when these events are activated. Crosscuts and Advice are integrated into a static scoping device called an Aspect that allows AOP programmers to conceptualise and integrate otherwise scattered changes to a system. Both the Advice and the crosscuts are language-specific mechanisms [3].

Recent studies on software evolution focus on dynamic evolution in distributed and heterogeneous system. Devanbu and Wohlstadter have proposed a multi-tiered, eclectic approach, and the design of the evolution specification language both draw from AOP languages [3, 2].

3. Proposed Approach

3.1 AOP with Microsoft .NET

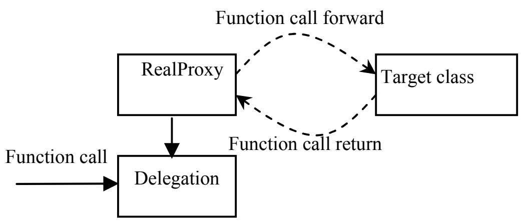


Figure 1. **Delegation Structure**

As in Figure 1, Microsoft .NET uses the base-class RealProxy as a delegation class. The Aspect functions will be implemented separately in each Aspect class and the instances of these classes will be linked in the message chain during the run time. Each object in the message chain transfers message to the next object by using SyncProcessMessage function. An abstract class, AspectAttribute, will be created as a base-class and all the Aspect classes will be derived from this abstract base-class.

All the messages, including constructor invocation, will be captured by the Delegation. For the first time, when the Delegation captures the message of constructor, the instance of target class will be created, and then the message chain can be constructed, in which the head node is the Delegation and the last node is the instance of target class. From this moment, all the messages will be transferred to the target object along the message chain.

A class that can be bound to a Context is called a context-bound class. A private Context object set up by the Microsoft .NET for an instance of a context-bound object provides the means for externally defined Aspects to hook into the message chain, because the creation of the private Context forces the creation of delegation class [5]. If an attribute class is derived from ProxyAttribute and used to decorate a context-bound class, then, the CreateInstance event of the attribute class can be triggered to construct the Delegation and establish the message chain before the creation of the instance of target class.

3.2 Analysis Rules

Not all the classes in the system are suitable for applying Aspect functions. Some rules are proposed to analyse the system and help the maintainer to make a better decision.

3.2.1 Rules for the Target Class Selection

Rule 1: *Aspect functions can only be applied to a class that can be derived from ContextBoundObject.*

Using this rule, the class, which is derived from a compiled class or COM object, cannot apply Aspect functions.

Rule 2: *Aspect functions can only be applied to a class that has no recursive public member functions.*

If there are recursive public member functions in the target class, the invocation of these functions will lead to too many checks in Delegation so that the system efficiency will be unbearable.

3.2.2 Rules for Joinpoint Selection

Rule 3: *If the public member function can be invoked before the creation of the instance of target class, it cannot be defined as a Joinpoint.*

If a member function, e.g., a static function, can be invoked before the creation of the instance of target class, it means that this function can be invoked before the creation of the Delegation, and accordingly, cannot be defined as a Joinpoint.

3.2.3 Rules for Benefits and Efficiency

Assuming: m = the number of target classes; k = the number of Joinpoints in one target class; l = the total line number of the source code, which is used to invoke the Aspect function; s = the total line number of the source code of Aspect function; δ = the total line number of the source code of Aspect base-class; the code conciseness rate, Cr, can be computed as:

$$C_r = \frac{m \times k \times l + s}{m + s + \delta} \tag{1}$$

$m \times k \times l + s$ represents the tangled code sizes in non-AOP-based implementation, $m + s + \delta$ represents the code sizes in AOP-based implementation.

Rule 4: *Cr should be higher than 1.*

Assuming: m = the number of target classes; Count_i = the number of public member functions of the ith target class. It should be noticed that the public property will be treated as a kind of special public member function; k = the number of Joinpoints in one target class; the system efficiency rate, E_r, can be computed as:

$$E_r = \frac{1}{\sum_{i=1}^{m} k \times Count_i} \qquad (2)$$

The value of denominator in formula (2) is the number of additional checks after applying Aspect function.

Rule 5: *If E_r is smaller than the low limit, the system efficiency cannot be accepted.*

The low limit depends on the computer environment and system configuration, which is still an estimative value drawn from static analysis of source code and will be different in varied applications.

4. Tool Support

Automation is one of the key goals of software evolution. The prototype tool, EvoWeaver, is a semi-automatic tool, which aims at helping software engineers in a comprehensive process of the AOP-based software evolution.

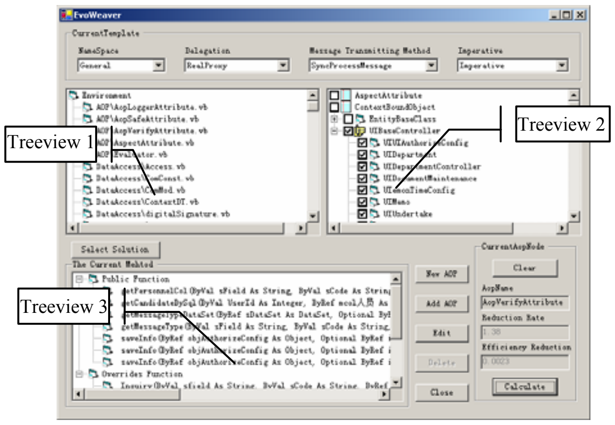


Figure 2. EvoWeaver Tool

Figure 2 shows the main window of EvoWeaver tool. Treeview 1 shows all the classes in the evolving system. Treeview 2 shows all the classes, which satisfy rule 1 and rule 2. If a class in the Treeview 2 is selected, rule 3 will be applied and all the properties and selected public member functions of this class will be shown in Treeview 3. After the target classes and Aspect functions are selected, the code conciseness rate and the efficiency rate of the evolving system can be calculated so that the maintainer can evaluate whether the evolved system is acceptable.

5. Case Studies

5.1 Example I: SQL Verification

Figure 3 depicts an existing software system [6], which needs to be evolved. The project is composed of one control class and several entity classes. The control class, DATAACCESS, is used to access the database system and invoked by several entity classes with the same method, e.g., SelectBySQL. A new task is required to add SQL verification function into some entity classes so that the SQL string can be verified before it is executed by database system. If there are suspicious or illegal characters in the SQL string, the operation will be interrupted.

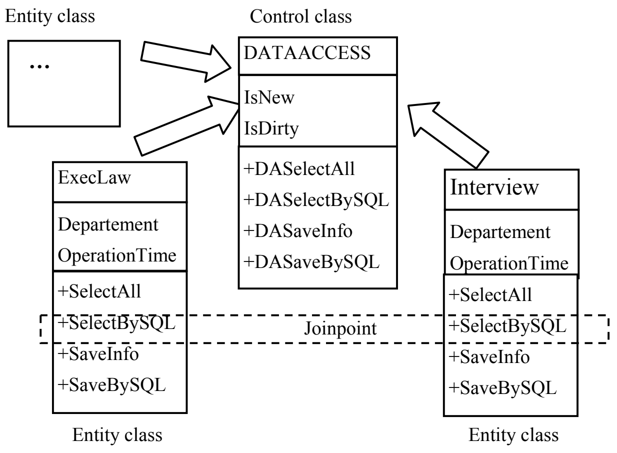


Figure 3. Class Diagram of SQL Verification

5.2 Example II: Event Logger

Event Logger is the function, which crosscuts all the entity classes and has been frequently talked in AOP. An Event Logger should track many elements, such as operator, operation and operation time, which can be gotten from the context in the event 'doAfterOperation', and record them into a log file. Event Logger is used to show the ability of continuous evolution with proposed approach, which new function can be inserted into the evolved system easily by adding the Logger attribute behind other attributes, like '<Verify(),Logger()>'.

5.3 Result Analysis

Table 1 is a general analysis of above two cases. The biggest benefit of AOP-based evolution is that the encapsulation of the classes and the structure of the system will not be destroyed and few interface checks need to be undertaken. It also shows that AOP technique can gain a positive outcome of concise code.

Table 1: Performance Analysis

	SQL Verification	Event Logger
Number of target classes	39	64
Joinpoints in each target class	1	6
Code line number without AOP	408	1824
Code line number with AOP	296	295
Rate of code conciseness	1.38	6.18
Decrease of interface checks	38	363
Additional judgements	439	5380
Efficiency rate	0.0023	0.0002

Many factors will influence the efficiency of a system, such as the number of Aspect classes, the number of the target classes and the number of the public member functions of each class.

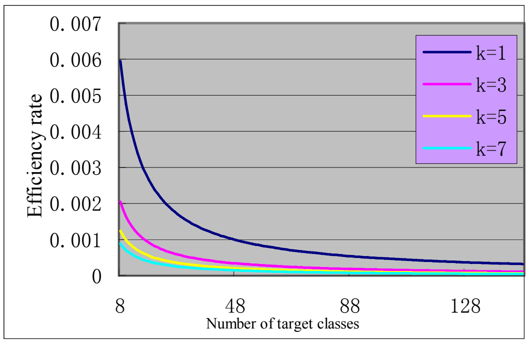


Figure 4. Efficiency Rate as a Function of the Number of Target Classes for Different k's Value

Figure 4 shows the efficiency rate as a function of m, the number of target classes, for various values of k. The conclusions can be drawn as follows:

1) The number of target classes has great influence on the system efficiency.

2) The efficiency rate decreases rapidly at first and remains almost constant after the number of target classes reaches 48.

3) When the number of Joinpoints (k in formula (2)) increases, the efficiency rate decreases rapidly.

6. Conclusions and Future Work

This study indicates that AOP technique is suitable for system evolution if the crosscut concern needs to be enhanced. New crosscut functions can be added to the system without destroying the system structure and encapsulation. With the proposed analysis rules, the prototype tool can help to apply Asepct functions, and evaluate the system benefits and efficiency.

Although a unified approach for AOP-based software evolution with Microsoft .NET has been presented, there are still issues to be addressed:

(1) Currently, the prototype tool can only analyse the system programmed in VB.NET. It should be enhanced to support more languages in .NET.

(2) The rules in EvoWeaver tool are still weak and should be strengthened.

(3) The low limit of the efficiency rate is still a empirically obtained value. Detailed rules should be given for different applications to get more exact values.

(4) There is still no mechanism to show how the system has been evolved and when the corresponding Aspect action can be preformed, as UML diagram does.

Finally, the Microsoft .NET supports the possibility of creating executing code at runtime, which is named as Dynamic Compile [12]. How to utilise this technique to build the Weaver and conquer the above disadvantages is still under research.

References

[1] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap", *The Future of Software Engineering*, ACM Press, New York, USA, 2000, pp. 75–87.

[2] Y. Coady, G. Kiczales, M. Feeley and G. Smolyn, "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code", *9th ACM SIGSOFT international symposium*, ACM Press, Vienna, Austria, 2001.

[3] P. Devanbu and E. Wohlstadter, "Evolution in Distributed Heterogeneous Systems", *Workshop on Reflection, AOP and Meta-Data for Software Evolution* ,Oslo, Norway, June 2004.

[4] P. Fradet and M. Sudholt, "AOP: towards a Generic Framework using Program Transformation and Analysis", *International Workshop on Aspect-Oriented Programming at ECOOP'98*, Brussels, Belgium, July 1998.

[5] E. Garson, "Aspect-Oriented Programming in C#/..NET", *Visual Systems Journal (VSJ)*, Bearpark, London, February 2004.

[6] H. Guo, F. Chen and Y. Wang, "A Reusable Software Architecture Model for Manufactory Management Information System", *26th IEEE International Conference on Computer Software and Application*, Oxford, England, September 2002.

[7] G. Kiczales, E. Hilsdale, et al, "An Overview of AspectJ", *European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001.

[8] G. Kiczales, J. Lamping, et al, "Aspect-Oriented Programming", *Proceedings of ECOOP'97*, Finland, June 1997.

[9] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, USA, 2003.

[10] N. Loughranl, A. Rashidl, W. Zhang and S. Jarzabek, "Supporting Product Line Evolution with Framed Aspects, *RAM-SE'04 ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution"* ,Oslo, Norway, 2004.

[11] S. L. Pfleeger, *Software Engineering; Theory and Practice (2nd Edition)*, Pearson Education, USA, 1998.

[12] G. T. Sullivan, "Aspect-Oriented Programming Using Reflection and Meta-Object Protocols", *Communications of the ACM*, 2001, Vol. 44, No. 10, pp. 95-97.

[13] D. Vollmann, "Visibility of Join-Points in AOP and Implementation Languages", *Second Workshop on Aspect-Oriented Software Development*, Bonn, 2002, pp. 65-69.

[14] J. Zhao, H. Yang, L. Xiang and B. Xu, "Change Impact Analysis to Support Architectural Evolution", *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, 2002, Vol. 14, pp. 317-333.

Requirements Engineering Challenges for Software Maintenance Projects in Distributed Software Development Environments

Rodrigo Santos de Espindola, Azriel Majdenbaum, Jorge Luiz Nicolas Audy
Programa de Pós-Graduação em Ciência da Computação
Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
{respindola, audy}@inf.pucrs.br , azm57@hotmail.com

Abstract

The growing market of distributed software development (DSD) creates new challenges for software maintenance and evolution. This kind of environment can increase the difficulties traditionally found on software maintenance, particularly in those ones that are related to Requirements Engineering (RE). The goal of this paper is analyze the challenges found on RE for software maintenance in DSD environments. For this purpose, it was made a case study on a multinational organization that uses DSD for maintenance of your legacy systems. This case study, of exploratory nature, made possible to identify the main difficulties found in this context. This case study, with exploratory nature, made possible to identify the main difficulties found in this context. As contribution for computer science, on software engineering area, this paper presents an initial proposal for the development of an approach for requirements management that it made possible to address the identified difficulties.

1. Introduction

The Software Engineering (SE) already has about forty years, but many of its first products, software systems developed in the decades of 60 and 70, continue being used until today. These systems, frequently called Legacy Systems, have high cost of maintenance and even small changes can bring problems for its maintainers [8].

Moreover, the development and the maintenance of computer-based systems come facing several difficulties in the last forty years. There are not a simple explanation for this phenomenon, but several studies point deficiencies in the systems requirements as one of the main causes of failures on software projects [2][3][4][5]. Such verifications have been taking some authors to consider RE as one of most important discipline of SE *[6]*.

In software maintenance projects several factors make the RE work more difficult, mainly the requirements management process. In DSD, factors like distance, communication and cultural differences contribute to make deeper the difficulties inherent to requirements management process, which acquires a still more critical character. However, taking into account the software maintenance importance and the growing adoption of DSD, still there are few studies about the impact they have on RE and, particularly, in requirements management process.

2. Problem characterization

According [11], with the increase of the business and with the changes on business process, several systems information become improperly in terms of capacity and functionality. These systems cannot be simply retired, they need to be improved and integrated to the organizational information infrastructure.

However, the lack of precision in the system documentation or, in the worst case, the non-existence of documentation and the original stakeholders unavailability make the work expensive and hard. This scenario is very common in practice, as pointed by several authors [10][6][12][7][11].

Another important factor to be considered is that, according [7], the RE is, usually, treated as an initial phase on software development. However, as show up by [7], the requirements knowledge requires a continuous effort in progressive refinement of the needs embedded in the organization business rules and on accommodating the stakeholders' needs, during all the software life cycle.

The RE theoretical bases of this paper was the works of [2], [3], [4], [5] and [6]. In software maintenance was used the works of [6], [7], [8], [10], [11] and [12]. It was not identified in the literature a study with the same focus proposed in this paper. It was found only related studies about RE in DSD where the focus was on software development project instead of software maintenance.

3. Case Study

The method used in this research was the case study, adopted as proposed by [9]. The teams involved in two software maintenance projects executed in a software development center located in Brazil had participated of this research. In project 1, both business analyst and final users are located in the USA. In project 2, business analyst and final users are located in a Brazilian branch office of the organization, located in another city.

With the accomplishment of the interviews and the application of the technique of content analysis were possible to identify the main difficulties found in the RE when it is applied on software maintenance project in DSD environments. Besides, it was possible relate the difficulties found in both empirical and theoretical studies.

After the analysis of the gotten answers and after the discussion among researchers, had been defined the following categories (Table 1):

Table 1. Identified difficulties

N°	Difficulty	Frequency
1	Lack of adequate documentation	5
2	Lack of knowledge in the application	4
3	Lack of methodology or standard for writing of requirements	4
4	Lack of planning	1

When questioned about the activities where these difficulties had been found, the majority of the respondents answered that the difficulties related to the requirements appeared in all the activities of the maintenance. Table 2 presents the identified categories.

Table 2. Affected activities

Activities	Frequency
All	5
Tests	2
Requirements Elicitation	1

The main difficulty pointed for the respondents, the lack of adequate documentation, confirm the theory, because diverse authors ([10] [11] [6] [12] [7]) point it as the main difficulty found in projects of maintenance of legacy systems. However, the DSD environment used in the unit adds new elements to this difficulty, because this situation also could be related to the difficulty of access to the documentation, caused for the geographic distribution between the team of maintainers and the others stakeholders of the project. Usually, the documentation used in projects developed in this kind of environment is divided among several distributed teams.

The lack of knowledge in the application, second difficulty pointed for the respondents, reinforces the theoretical study, once it is directly related to the unavailability of the original stakeholders. Besides, in DSD environments, factors as cultural and language differences make difficult the process of application knowledge transfer among the stakeholders geographically distributed.

Another difficulty cited for the respondents was lack of methodology or standard for writing of requirements in maintenance projects. The requirements currently are written in natural language, without the use of none formal technique of specification.

Only one of the respondents pointed the lack of planning as a difficulty faced in the RE. This respondent believes that the problems could be minimized if it had more planning to address to the requirements and the modifications in the requirements.

3.1. Critical analysis of results

With the results of the case study it was possible to get relevant information as much to the requirements engineering process on software maintenance projects in DSD environments and the main difficulties faced for the respondents.

One important aspect observed during the study is related to the requirements management process in the maintenance projects. This difficulty is related in part with the lack of proper documentation that allows the recovery of the original requirements of the systems submitted to the maintenance and the correct record of the changes made in these requirements. Being the difficulties relative to the documentation of the system can be aggravated in DSD environment where the projects are accomplished. Therefore that the documentation containing the requirements of the system exists, it can be in another site of development, or writing in another language and context. It makes difficult to maintain traceability information. Moreover, in DSD the requirements need to be created or updated in different sites, for different stakeholders.

Once more, the case study results had confirmed the theory, showing the practices that bond requirements to the project scope, instead of software product scope that is being developed or maintained.

This approach leads to a fragmentation of the requirements process and the requirements specifications, such as illustrated for Figure 1. For each project the requirements engineering process (R.E.) is executed again, aiming at the creation of the requirements document contemplating the needs for the maintenance that must be executed. Soon after this the requirements management process is executed (R.M.), contemplating only the requirements changes occurred in the project on this last created document. This fragmentation of the requirements specifications contributes for the difficulties of attainment of legacy application knowledge, such as pointed by the respondents.

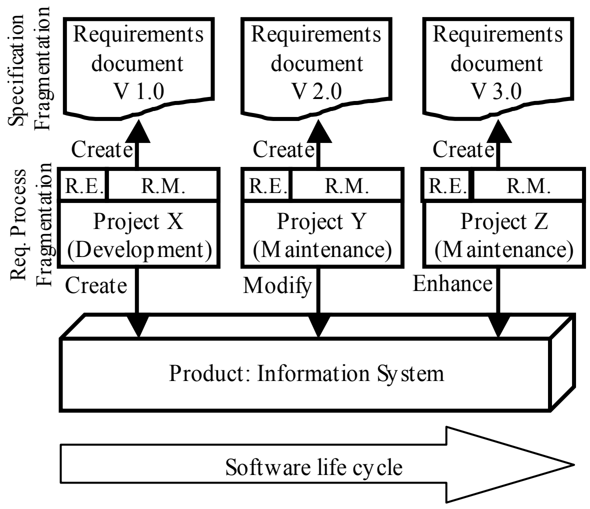

Specification Fragmentation
Requirements document V 1.0
Requirements document V 2.0
Requirements document V 3.0
Create
Create
Create
Req. Process Fragmentation
R.E.
R.M.
Project X (Development)
Project Y (Maintenance)
Project Z (Maintenance)
Create
Modify
Enhance
Product: Information System
Software life cycle

Figure 1. Fragmented requirement process.

In the bibliographical research accomplished, relevant contributions had not been found in literature that allows solving the difficulties identified in this case study. The contributions that are more closed to the context of this study are limited to the requirements engineering process in DSD environments on software development projects, lacking of mechanisms that deal with the reality found in the legacy systems maintenance.

4. Proposed approach

To contribute for the solution of the difficulties explained in sections 2 and 3, this paper presents initial proposal for development of an approach for requirements management process for maintenance in DSD. The basic idea behind of this proposal consists of an integrated requirements management process that prevents the fragmenting of the requirements specifications.

In [5] it is presented an approach of RE focused on software development projects where stakeholders are co-located. The requirements management process proposed here, aims at to adapt this approach for creation of a requirements process for DSD that contemplates the complete software life cycle, which includes the software development and its evolution throughout several maintenance projects. Figure 2 illustrates the integrated nature of the requirements management process and the context where this is inserted.

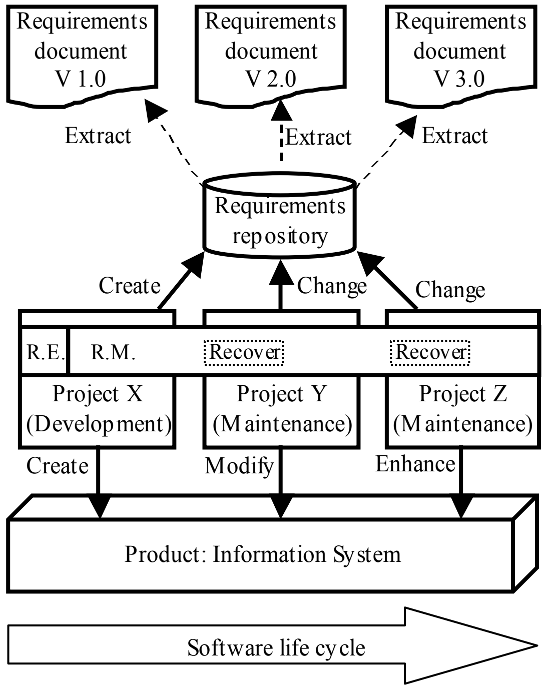


Figure 2. Requirements process integrated to the software life cycle.

In this approach, the requirements engineering process continues being used in the beginning of the product life cycle. The difference is in its objective. Instead of creating the requirements document for each project, this process aims at creates a requirements repository for the software product.

This characteristic of the proposed approach aims to change the requirements engineering focus to create and maintain only one requirements specification through all software product life cycle, independently of the stakeholder's location or the several maintenance projects to be developed. This way, this work intends eliminate the difficulties caused by adoption of practices that bond the requirements to the project scope, instead of software product scope that is being developed or maintained, as presented in sections 2 and 3.

This work intends also to reduce the impact of difficulties caused by the lack of original stakeholders and the lack of application knowledge, as presented in sections 2 and 3. This is due the fact that the maintenance of a complete and updated requirements specification can reduce the need of asking help to alternative sources to recovery the information about the original software requirements before each maintenance project. This way, the maintainers can obtain the necessary knowledge in original and current application requirements before starting the maintenance work.

Hence, the requirements repository is a key part of this approach and can be implemented in a database management system (DBMS) or using a tool specialized on software requirements, such as RequisitePro or DOORS.

After the requirements repository creation and the initial baseline of requirements approval, the requirements management process starts to be executed. This process keeps the original purpose, that is, to manage the software requirements changes. The difference is that in this approach the process is not restricted to the system development project, but continues being executed throughout all the software life cycle. With this approach it expects to maintain the requirements specification up to date and to guarantee a source of information that allows improving the knowledge in the legacy system, thus addressing the two main difficulties identified in both case study and theory.

In addition, this process has an optional activity for requirements recovery. This activity must be executed when the legacy system not has a requirements repository. In this case, reverse engineering and requirements recovery techniques must be used, such as AMBOLS [11] e CelLEST [1], for the creation of the repository, being another point of flexibility of the process. The inclusion of this activity aims at to address to the difficulties caused for the typical scenario of legacy systems maintenance, where it does not exist proper documentation, as was already presented in sections 2 and 3.

5. Final Considerations

The RE in DSD comes to stimulating increasing interest in the academic community. However, relevant studies about this subject are still not found in legacy systems maintenance area. This is a context particularly problematical to the RE, considering that the difficulties usually found on software maintenance and evolution areas tends to be aggravated when this work must be done in a geographically distributed way.

As the main contribution this paper presents an initial proposal for the development of an approach for requirements management in DSD environments. Finally, this study aims to contribute with practical when taking care of an increasing organizational demand for improvements in the RE processes, as well as, for dealing with difficulties faced in the software maintenance in DSD environments.

6. References

[1] El-Ramly, M., Stroulia, E., Sorenson, P. Recovering Software Requirements from System-user Interaction Traces. In Proc. SEKE'02, Ichia, Italy, July 2002, ACM, pp.447-454.

[2] Standish Group. "CHAOS Report". Captured in: http://www.standishgroup.com , 1995.

[3] Sommerville, I., Sawyer, P. Requirements Engineering – a good practice guide. John Wiley & Sons Ltd, New York, 1997.

[4] Leffingwell, D., Widrig, D. Managing Software Requirements – A Unified Approach. Addison-Wesley. 2000.

[5] Kotonya, G., Sommerville, I. Requirements Engineering: process and techniques. John Wiley & Sons Ltd, New York , 1998.

[6] Pressman, R. S. Software Engineering: a practitioner's approach. McGraw Hill, New York, 5th ed., 2001.

[7] Zanlorenci, E. P., Burnett, R. C. "Abordagem de Engenharia de Requisitos em Software Legado". Workshop em Engenharia de Requisitos, Piracicaba-SP, Brasil, 2003, pp 270-284.

[8] Lucia, A., Fasolino, A. R., Pompella, E. "A Decisional Framework for Legacy System Management". International Conference on Software Maintenance, 2001.

[9] Yin, R. Estudo de Caso: planejamento e métodos. Bookman, São Paulo, 2001.

[10] Ebner, G., Kaindl, H., "Tracing All Around in Reengineering", IEEE Software, May 2002, pp.70-76.

[11] Liu, K., Alderson, A., Qureshi, Z., "Requirements Recovery from Legacy Systems by Analysing and Modelling Behaviour". Proceedings of the International Conference on Software Maintenance, 1999, IEEE Computer Society, Los Alamitos, pp3-12.

[12] White, Stephanie M. "Capturing Requirements for Legacy Systems". Proceedings of the International Symposium and Workshop on Systems Engineering of Computer Based Systems, 1995. pp 251-256.

Automatic Detection of *Bad Smells* Using Software Metrics

Beatriz Florián
Universidad de los Andes
befloria@yahoo.com

Ángela Lozano
Universidad de los Andes
ang-loza@uniandes.edu.co

Silvia Takahashi
Universidad de los Andes
stakahas@uniandes.edu.co

Abstract

When dealing with the maintenance of legacy software systems, we must start by fixing structural problems with evolutionary and safe changes. These initial changes are mere refactorings that do not change the program's functionality. However, we must first detect the parts of the code that present problems or bad smells, as they are commonly known. .

This paper deals with this detection problem. We propose that certain metrics which can be computed statically can be used to detect bad smells.

***Keywords**: Reengineering, maintenance, refactoring, bad smells*

1 Introduction

Refactoring is a technique used to enhance an application's maintainability by improving its internal design and making the source code easier to understand. Changes do not change the application's functionality. One of the most difficult issues in refactoring is the identification of which parts of the source code need refactoring.

This paper summarizes the results of Beatriz Florian's Master's Thesis [11]. It is the first phase of a larger project that aims to develop a methodology for detecting bad smells.

The rest of this paper is organized as follows. Section 2 gives a brief summary of the state of the art in refactoring, particularly in what pertains to bad smells. In Section 3, we describe the bad smells that we studied in our research and outline and our approach for automatically detecting them. In Section 5, we describe how we tested our approach and we present the results we obtained. Finally, Section 6 presents some conclusions and directions for future research.

2 Background

The most well known is the work of Fowler [1] who enumerates a set of patterns, called bad smells that may represent a source code flaw. However, he emphatically states that human intuition cannot be replaced for detecting bad smells. Many other authors have attempted to find ways to detect bad smells automatically.

One of the most important contributions to the study of bad smells is the taxonomy proposed by Wake [2]. This guided the development of the techniques that we propose in this paper.

Other authors have attempted to define mechanisms for detecting bad smells automatically. The most significant papers on which we based our work were: [3], [8], [7], [6], [10], and [4].

3 Detecting Bad Smells for Refactoring

We deal four of the groups of bad smells from Wake's taxonomy [2] (See Table 1).

Grupo	Bad Smell	Defined	Implemented
Measured Smells	*Long Method*	✓	✓
	Large Class	✓	✓
	Long Parameter List	✓	✓
	Comments	✓	
Duplication	*Duplicated Code*	✓	
Data	*Data Class*	✓	✓
	Data Clump		
	Primitive Obsesión		
Unnecessary Code	*Lazy Class*	✓	
	Speculative Generality	✓	
	Temporary Field	✓	

Table 1: Bad Smell Detection

One of the contributions of this paper is to present a uniform structure to describe the approach for the detection of each bad smell.

We propose a worksheet that should be completed for each bad smell. The worksheet includes the following information: Name, Description, Motivation, Proposed strategies for detection, and Metrics used for each strategy. If there is more than one strategy, then it states which one is chosen. Finally, the criteria, which will indicate whether or not the bad smell is present, must be described. The rationale used to choose the strategy and the criteria used to determine whether or not a bad smell is present is also included.

The rest of this section includes the worksheets for some of the bad smells we studied.

3.1 Large Class

Description*:* Large classes that attempt to carry out too many tasks or that have many attributes. Large classes usually have a many attributes or methods. Maintenance is difficult. Program understanding is also affected.
Motivation: Improve maintainability and understanding.
Strategies and Metrics:

Strategy	Metrics
Detect classes that have many methods and attributes.	• Number of Fields (NOF), • Number of Methods (NOM)
Measure the cohesion of a class.	• Lack of Cohesion Methods (LCOM)
Determine the weight of a class in terms of the weight of its methods.	• Number of Methods (NOM), • Weighted Methods/Class (WMC).
First we detect classes that depend strongly on light classes without considering small cohesive classes [8].	• Access Of Foreign Data (AOFD), • Weighted Method Count (WMC), • Tight Class Cohesion (TCC).

Table 2: Strategies and Metrics for Large Class

Selected Strategy: We chose the third option.
Criteria: WMC > 10*NOM.
Rationale: Use WMC to measure a class' complexity. This metric will also be used to determine if a class has too much responsibility. The accepted range for CC is between 1 y 10 (see[10]). If WMC is determined using CC, the sum of complexities for WMC, should be less than 10 times the number of methods in the class. If the average of the complexities of the methods is greater than 10, the class has many alternate paths and can be perceived as a large class.

3.2 Long Method

Description: A very long method which is difficult to understand, to change, and to extend.
Motivation: Long methods can be decomposed to improve clarity and ease maintenance.
Strategies and Metrics:

Strategy	Metrics
Detect an excessive number of statements and temporal values in a method.	• (lLOC): Number of Statements • (NOTM): Number Of Temporal values of Method
Measure number of statements and also the complexity.	• (lLOC) Number of Statements • (VG) McCabe´s Cyclomatic Complexity
Measure complexity using the weight of a class' methods.	(WMC) Weighted Method Count

Table 3: Strategies and Metrics for Long Method

Selected Strategy: We chose option 2.
Criteria: ((VG > 10) && (lLOC > 20)).
Rationale: Though option 3 also measures method complexity, option 2 also considers the number of declarations; we feel that option 2 is better. Option 2 is also better than option 1 because it measures complexity instead of temporary values. To determine the criteria we used the following rationale: Both metrics should be above the accepted values defined in [10].

3.3 Long Parameter List

Description: Long parameter lists often make methods difficult to understand. It may indicate that classes are not well defined.
Motivation: This problem can be fixed by encapsulating various parameters in one class. By doing this, the overall design of the application can be improved. By reducing long parameter lists, we obtain methods that can be understood and maintained easily.
Strategies: In this case, we propose only one strategy with its corresponding metric and criteria. Many authors only consider the number of parameters as an isolated value. In fact, some authors (see [10]) have determined that the number of parameters should be no more that four. We do not believe that the number of parameters is enough. We must relate it to the number of statements in the method.

Metrics: To implement the option, we need two metrics: number of parameters (PAR) and number of statements (ILOC).
Criteria and rationale: When (PAR > (0.2) ILOC), the bad smell should be detected. Though 20%, was determined intuitively, in the testing phase we were able to compare it with other values and confirmed that it provided a high level of accuracy.

3.4 Data Class

Description: Classes that only contain data with no logic other than methods to get values and modify attributes.
Motivation: Program Understanding.
Strategy: Find light classes that provide few services and only have sets and gets. In Java programs, declaring public fields is also a bad smell [8].
Metrics:

- (WOC): Weight of a classs
- (NOPA): Number of public attributes
- (NOAM): Number of accessing methods
- (NOM): Number of methods

Criteria:
$(WOC < 2/3) \vee (NOAM > 2/3NOM) \vee (NOPA < 2/3NOM)$
Rationale: The first two conditions set a limit on the percentage of getters and setters in comparison to the other methods. The third condition is used to determine the presence of public fields.

3.5 Speculative Generality

Description: Many times programmers include code that may be used in a later time. There are methods, attributes, or parameters that are not being used.
Motivation: Code like this is often difficult to understand. By removing unused variables and methods we improve the code's overall structure.
Strategy: Detect unused methods and attributes. Detect abstract methods that are not defined.
Metrics:

- (UNV): Unused variables
- Unused Method
- Unused Parameter
- Abstract method not implemented

Criteria: In this case, we decided to be very strict. If any of the metrics is greater than zero, we detect the bad smell.

3.6 Lazy Class

Description: A class that does not have much responsibility. Many times its methods can be included in another class.
Motivation: By removing lazy classes we can improve program understanding.
Strategy: Find light cohesive classes.
Metrics:

- (WMC): Weighted Method Count
- (TCC): Tight Class Cohesion
- (NOM): Number of Methods

Criteria:
$((WMC < NOM*2) \vee (TCC > (0.2)(NOM/2)))$.
Rationale: The first condition indicates that the sum of complexity is low because, in average, it is less than two paths per method. The second condition indicates that the number of methods that are directly connected is greater than 20%.

3.7 Comments

Description: Though uncommented code clearly is a bad software practice, comments can also be used to compensate for a defective software structure.

Motivation: By eliminating unnecessary comments where these indicate design errors, the overall structure of the application can be improved.

Strategies, metrics and criteria: In this case, we also propose only one strategy: we detect the percentage of comments in the code. There is a metric for this purpose: comment percentage (CP). In [10], it states that an appropriate value for this metric is 30%. We use the same criteria.

3.8 Duplication

Description: This is one of the worst smells. It occurs when the same code structure appears many times throughout the code. This duplication can be syntactic or semantic. It is easier to detect when it is syntactic duplication.

Motivation: Improve software understanding and maintainability. Many times when we have duplicated code and it has to be modified, it will have to be modified everywhere it appears.

Strategies, metrics and criteria: We propose only one strategy: use the abstract syntax tree to detect syntactic duplication. We propose to carry out this analysis by phases: within the same method; among methods of the same class and finally among methods of the same class hierarchy. There is no metric for this purpose, so we propose a new one: percentage of duplicated code in a class. Since this one of the worst patters of bad software coding we believe that only 10% duplication should be allowed. We did not implement this strategy because of time constraints. Therefore, we cannot be sure whether or not the 10% value is adequate.

4 Tests and Results

We developed an Eclipse plug-in that obtained data from other tools that compute metrics ([12], [13]).

To test our technique we chose a large application [9]. This application has 14 packages and 70 classes. We used our tool to detect bad smells and compared it to results obtained by two programmers. When evaluating methods we studied 176 methods from 10 different classes. The tables below show the results we obtained. In Table 4, we show the number of bad smells detected by our tool and the ones detected by at least one of the human subjects. Table 5 shows the percentage number of matches (where both the human subjects and the detection tool detected the bad smell); false positives (where a bad smell was detected by the tool but not by the human subject); and false negatives (detected by the human subject, but not the tool).

Bad Smell	Human Detection	Automatic Detection
Large Class	22	26
Long Method	7	6
Long Parameter List	4	4
Data Class	6	8

Table 4: Human vs Automatic Bad Smell Detection

Bad Smell	Matches	False Positives	False Negatives
Large Class	50.00%	31.25%	18.75%
Long Method	44.44%	22.22%	33.33%
Long Parameter List	60.00%	20.00%	20.00%
Data Class	75.00%	25.00%	0.00%

Table 5: Matches, False Positives and False Negatives Percentiles

We can see that there is a high incidence of matches for all the bad smells we studied which seem to suggest that automatic detection can compete with human intuition. The false positives can be due to the fact that the human subjects incorrectly missed the bad smell. We need to incorporate more human test subjects for our future research. These values are acceptable for the bad smells Data Class and Long Parameter List.

5 Conclusions and Future Research

This paper summarized the results of a research project in which techniques for detecting some of the well-known bad smells using metrics were designed, developed and tested. In comparison with the work of other authors, our proposal addresses more bad smells and it is one of the few ones that deal specifically with Java code.

Future research should deal with the detection of all the bad smells that were described in this project. This, of course, leads to the characterization and implementation of detection strategies for other bad smells. The tool should allow the user to change the criteria so that it can be adjusted according to the results.

6 References

[1] M. Fowler. “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, 1999

[2] W. C., Wake. “Refactoring Workbook”. Addison-Wesley, August 2003

[3] F. Muñoz Bravo. “A Logic Meta-Programming Framework for Supporting the Refactoring Process”. Thesis of Master of Science in Computer Science, University of Brussel. 2003.

[4] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” International Conf. Software Maintenance, pp. 109-118, 1999.

[5] T. Tourwé, T. Mens, “Identifying Refactoring Opportunities Using Logia Meta Programming”. Proc. European conf. Sortware Maintenance and Reeng., pp. 91-100, 2003.

[6] E. Van Emden and L. Moonen, “Java Quality Assurance by Detecting Code Smells”. Proc. Working Conf. Reverse Eng., pp. 97-108, 2002.

[7] F. Simmon, F. Steinbrückner, and C. Lewerentz, “Metrics Based Refactoring”. Proc. European Conf. Software Maintenance an Reeng., pp. 30-38, 2001

[8] R. Marinescu. “Detecting Desing Flaws Via Metrics In Object Oriented Systems”.

[9] Fernando Solano Donado, http://bcds.eia.udg.edu.es/fsolanod Universidad de Girona, 2004

[10] http://www.refactorit.com/?id=29605. Metrics Table and References.

[11] Beatriz Florian, Detección de Bad Smells en Aplicaciones JAVA Utilizando Métricas de Software, Master’s Thesis, Universidad de los Andes, Bogotá, Colombia, 2005.

[12] http://www.teaminabox.co.uk/downloads/metrics/versions.html TEAMINABOX Eclipse Metrics Plugin

[13] http://www.sourceforge.net/projects/metrics. Source Forge Eclipse Metrics Plugin

REGoLive: Adding Web Site Comprehension to Adobe GoLive

Grace Gui, Holger M. Kienle, and Hausi A. Müller
Computer Science Department
University of Victoria, Canada
{gracegui,kienle,hausi}@cs.uvic.ca

Abstract

This paper describes a demonstration of the REGoLive reverse engineering tool. REGoLive extends Adobe GoLive with sophisticated reverse engineering functionality for Web site comprehension. This functionality was realized via programmatic customization of GoLive with JavaScript, and the use of Web services to communicate with our SVG graph visualization engine. The paper explains how we implemented REGoLive, and addresses why leveraging of popular off-the-shelf components such as GoLive has a number of potential benefits from the user's perspective.

1. Introduction

The reverse engineering and program comprehension community has developed many tools to understand better complex software systems. Nowadays, many Web sites are in fact highly complex software systems [6]. This has caused the emergence of Web site reverse engineering, which proposes to apply reverse engineering approaches to the domain of Web sites. There are a number of research tools that help Web site comprehension (e.g., [9] [8] [2]).

Traditionally, program comprehension functionality is implemented with stand-alone tools. As a result, software engineers typically have to switch between various tools during comprehension activities. Each of these tools has its own idiosyncratic user interface and interaction paradigm, causing an unfavorable learning curve. As a result, many program comprehension tools fail to be adopted. Software engineering activities that involve program comprehension (e.g., maintenance) require the use of forward engineering tools (e.g., compilers) as well as reverse engineering tools (e.g., class hierarchy visualizers). Thus, extending forward engineering tools such as IDEs (e.g., Eclipse) or Web authoring tools (e.g., GoLive) by seamlessly adding program comprehension functionality helps software engineers and improves the adoption of comprehension functionality [5]. REGoLive is an example of an adoption-centric tool development approach that leverages an existing popular Web authoring tool, GoLive, by grafting functionality for Web site comprehension on top.

2. GoLive

We chose to use GoLive as host product for our tool implementation because of its popularity, maturity, and extensive customization support. GoLive is a mature product that has evolved through several major releases; currently we are using Version 6.0.

GoLive already provides rudimentary support for Web site comprehension activities for redocumentation, program analysis, data gathering, knowledge management, and information exploration. Information in GoLive is presented to the user with views. There are a large number of views, showing various properties of the Web site. The *Files view* lists the files (e.g., pages, images, and scripts) belonging to a Web site. Some views focus on a single page (e.g., Source Code Editor and Layout Preview), while others show relationships between pages (e.g., In & Out Links and Navigation). Whereas GoLive offers information exploration with views, it has no graph visualization, which is the preferred visualization of most program comprehension tools. As a result, information in GoLive is dispersed over several views. However, it would be desirable to have a complementary graph visualization providing a unified view of a Web site, and allowing sophisticated manipulations such as building of hierarchies.

The GoLive Extend Script Software Developer's Kit (SDK) enables programmatic customization via so-called Extend Scripts. The SDK provides numerous JavaScript objects and methods to programmatically manipulate files and folders as well as the contents of documents written in HTML, XML, JSP, etc. The document content that has been read into memory is made available in GoLive through a Document Object Model (DOM), which allows to query and to manipulate markup elements. Thus, batch processing of changes to an entire Web site can be easily accomplished.

3. REGoLive

Program comprehension tools are usually handcrafted and stand-alone. Our tool-building approach for REGoLive is different, because we are leveraging an existing product and augmenting it with program comprehension functionality. When doing this, we can take advantage of the (program comprehension) functionality already offered by the host product, focusing on the missing pieces. As a result, GoLive users engaged in program comprehension can seamlessly transition back and forth between GoLive and REGoLive functionality.

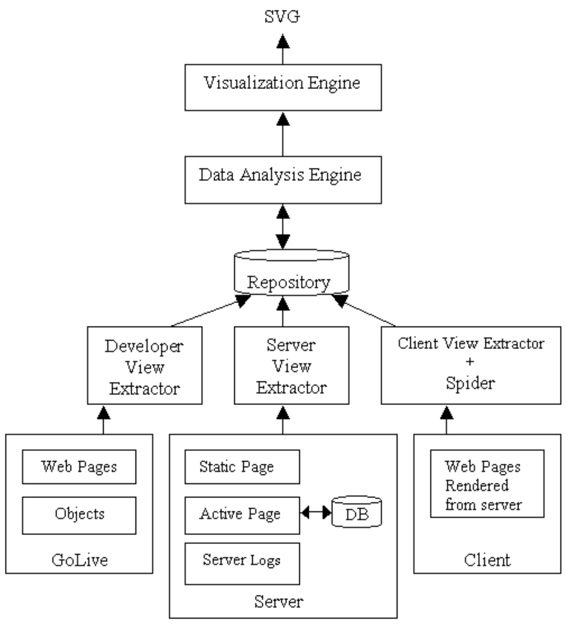


Figure 1. Architecture of REGoLive

The architecture of the REGoLive tool is depicted in Figure 1. REGoLive allows the reverse engineer to comprehend a Web site with three distinct viewpoints [1]: the developer view, which shows the Web site as a developer sees it (by using a Web authoring tool such as GoLive); the server view, which is the result of telling the Web authoring tool to deploy the site on a Web server; and the client view, which represents the Web site as a user sees it (by using a Web browser).

In order to support the three viewpoints, different fact extractors for each view had to be written. The developer-view extractor retrieves information that is provided by GoLive about the currently loaded Web site (cf. left tree-view in Figure 3). The artifacts in the developer view include files (such as Web pages, CSS files, and JSPs) as well as tool-specific objects (such as templates and smart objects). The server-view and client-view extractors work on their respective viewpoints, extracting similar artifacts. All extractors are implemented as extensions to GoLive and written in JavaScript.

The extractors write the extracted facts into a repository, which is currently implemented as a flat file in GXL format [3]. Then data analysis constructs the mappings between different views. These mappings show how artifacts in one view are related to artifacts in other views.

The visualization engine presents the result of the data analysis to the reverse engineer. Figure 2 shows a screenshot of the developer view of a Web site. Artifacts are visualized as nodes in a graph editor (e.g., blue nodes represent HTML pages while yellow nodes represent GoLive templates). Relationships between artifacts are shown as arcs between nodes (e.g., a yellow arc from a blue node to a yellow node indicates that a HTML page makes use of a template). The graph is rendered with Scalable Vector Graphics (SVG) [7] in a Web browser. We currently use Adobe's SVG viewer to render the SVG in Internet Explorer. The SVG graph editor allows interactive exploration of the graph, including moving of nodes, filtering of arcs and nodes, searching, and applying graph layouts. The graph editor has been implemented in JavaScript and is a separate component that can be customized for different domains [4]. Control integration between REGoLive and the SVG graph editor is achieved via Web services, which allow us to send messages between the two components. For example, selecting a graph node in SVG sends a message to GoLive to select the corresponding entities in the views.

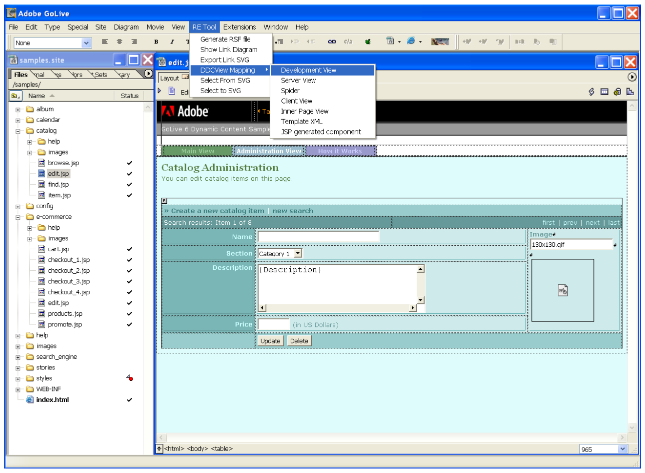


Figure 3. REGoLive adds a drop-down menu to GoLive to access the viewpoints of the currently active Web site

A reverse engineer can call up the different views from an extra pull-down menu ("RE Tool") in GoLive. Figure 3 shows a screenshot of the added menu. It is also possible to

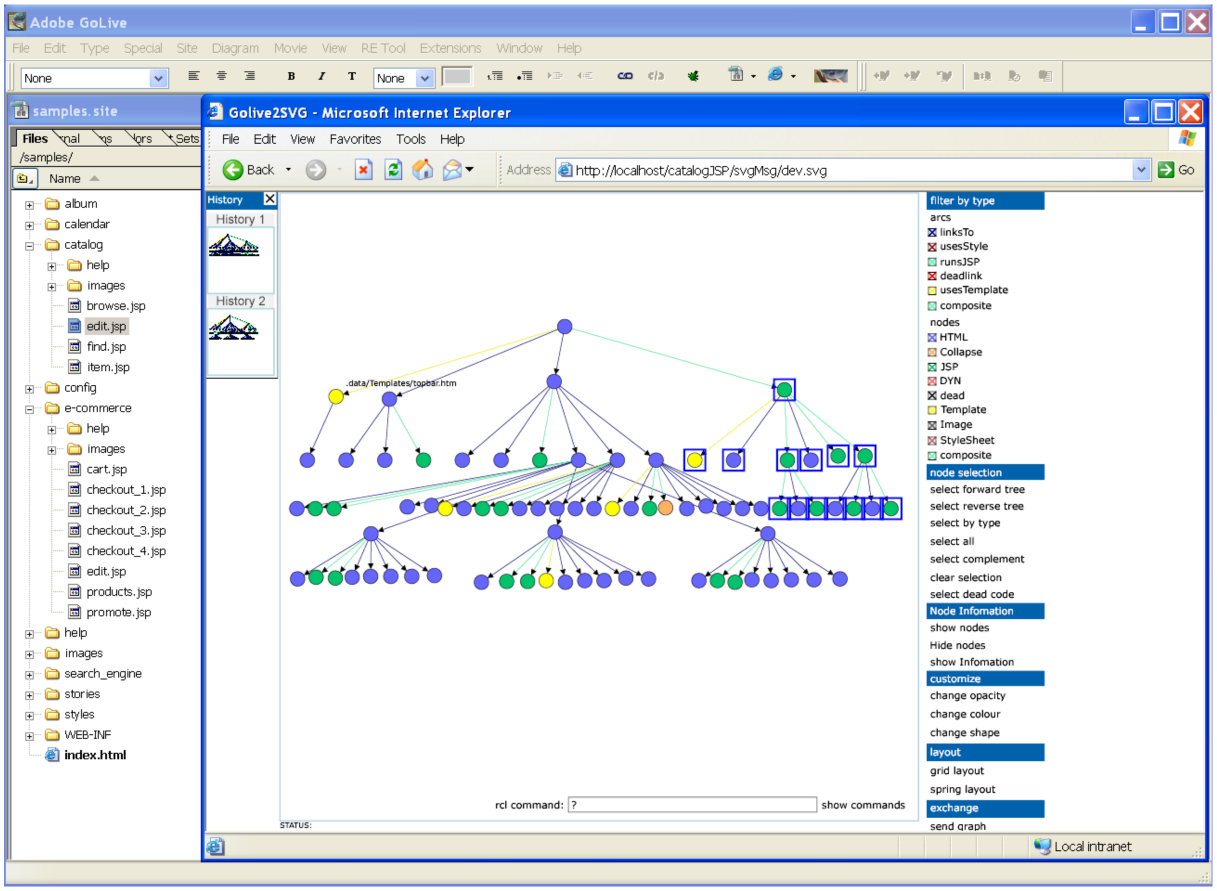


Figure 2. REGoLive's developer view

navigate from nodes shown in the SVG graph editor to the corresponding artifacts in GoLive (and vice versa).

REGoLive can be seen as a proof-of-concept of our tool-building approach, which strives to make reverse engineering tools more adoptable by building new program comprehension functionality on top of existing, popular off-the-shelf products.

Acknowledgments

This work has been supported by the IBM Toronto Center for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Consortium for Software Engineering (CSER).

References

[1] G. Gui, H. M. Kienle, and H. A. Müller. REGoLive: Web site comprehension with viewpoints. *13th IEEE International Workhop on Program Comprehension (IWPC'05)*, May 2005.

[2] A. E. Hassan and R. C. Holt. Towards a better understanding of web applications. *3rd International Workshop on Web Site Evolution (WSE 2001)*, pages 112–116, Nov. 2001.

[3] R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. *Seventh Working Conference on Reverse Engineering (WCRE '00)*, pages 162–171, Nov. 2000.

[4] H. M. Kienle, A. Weber, and H. A. Müller. Leveraging SVG in the Rigi reverse engineering tool. *SVG Open / Carto.net Developers Conference*, July 2002.

[5] H. A. Müller, A. Weber, and K. Wong. Leveraging cognitive support and modern platforms for adoption-centric reverse engineering (ACRE). *3rd International Workshop on Adoption-Centric Software Engineering (ACSE 2003)*, pages 30–35, May 2003.

[6] J. Offutt. Quality attributes of web software applications. *IEEE Software*, 19(2):25–32, Mar./Apr. 2002.

[7] A. Quint. Scalable vector graphics. *IEEE MultiMedia*, 10(3):99–101, July–Sept. 2003.

[8] F. Ricca and P. Tonella. Web site analysis: Structure and evolution. *International Conference on Software Maintenance (ICSM '00)*, pages 76–86, Oct. 2000.

[9] P. Warren, C. Boldyreff, and M. Munro. The evolution of websites. *7th International Workshop on Program Comprehension (IWPC '99)*, pages 178–185, 1999.

Using Program Slicing Metrics to Predict the Maintenance of Software

Tracy Hall and Paul Wernick

Systems and Software Group, School of Computer Science
University of Hertfordshire
College Lane, Hatfield, Hertfordshire AL10 9AB, England
tel. ++1707 286323/284782; fax ++1707 284303
{t.hall, p.d.wernick}@herts.ac.uk

Abstract

Previous research has identified a number of metrics derived from program slicing. In this paper we discuss how these metrics relate to the effort required to maintain an existing software-based system. Whilst our interest in this work stems from our development of simulation models of long-term software maintenance processes, it will also be directly relevant to the managers of software maintenance activities.

Keywords: software maintainability, program slicing, metrics, simulation

1. Introduction

In this paper we investigate how program slicing metric data can be used to measure the maintainability of software systems. We suggest that values for program slicing-based metrics, used in combination with size data, can assist in the prediction of the maintainability of systems over time. This extends our work on modelling and predicting long-term software maintenance trends [7].

Historically measuring maintainability has been performed rather unsatisfactorily. There are currently no generally accepted measures for the maintainability of systems. There has been little input from any underlying theory of software maintenance in the derivation of current metrics which would allow them to be related to the actual *maintainability* of a software system.

In our previous work we have simulated the long-term maintenance of software systems using system dynamics [7]. In this work, we found that the difficulty of quantifying the maintainability of a system at any particular time, and changes in that value over time, became a major issue. The lack of metrics which can plausibly reflect the ease or difficulty in maintaining an existing software system made that part of our simulation difficult to quantify. As a result, we found it difficult to predict with confidence the impact of a process change on the long-term maintenance of a system.

In order to capture the effect of the existing system on further changes to it, we have developed the notion of 'inertia'. We define inertia as an indirect measure of maintainability which has two dimensions: change (usually growth) in the size of the system as it is evolved, and change to the structure and code of the system as it is maintained. Growth in system size may make the system correspondingly more difficult to maintain. However, size alone does not capture the full richness of inertia, since two systems of equal size may not be equally maintainable.

Meyers and Binkley's work [3] on program slicing-based metrics provides a possible approach to addressing this issue. Meyers and Binkley have conducted longitudinal studies into the behaviour of a number of the metrics described by Weiser [8] and by Ott and Thuss [5]. The use of slicing-based metrics has been proposed previously to focus maintenance interventions and direct re-engineering effort. In this paper we describe an alternative application of slicing data, in which we use these metrics to help quantify the maintainability of software systems, rather than using them as an aid in re-engineering systems.

This paper addresses two research questions:

1. Are slice-based measures a viable approach to generating data whose values and trends characterise maintainability?

2. Can maintainability data contribute to predicting the long-term maintenance of software systems?

2. Slicing Metrics

Program slicing was first proposed by Weiser [8, 9] as a technique to assist in debugging programs. The idea emerged in response to Weiser's observations on how experienced debuggers find faults in programs. In its simplest form program slicing identifies all parts of a program that are related to a given statement. This means that all statements that do not affect a particular variable at a specific point in the program are removed. The resulting partial program is referred to as a 'program slice'.

A number of metrics have been proposed to describe the program slices which can be identified for a system. Slicing-based metrics were first described by Weiser [8] and then extended in the early 1990s by Ott and Thuss [5], in order to characterise the slices which they obtained. Metrics originally proposed by Weiser [8] are described in Table 1. Two further metrics proposed by Ott and Thuss [5] are presented in Table 2.

More recently, tools have become available which allow the collection of larger-scale slicing data. Meyers and Binkley were the first to collect and analyse such larger-scale data [3]. However the potential for using slicing data in relation to subsequent releases of systems has long been recognised. Ott and Thuss [4] suggested the need for such work.

Table 1. Slicing-based metrics proposed by Weiser [8]

Metric	Description
Coverage	Compares the length of slices to the length of the entire program. Coverage might be expressed as the ratio of mean slice length to program length. A low coverage value, indicating a long program with many short slices, may indicate a program which has several distinct conceptual purposes.
Overlap	Is a measure of how many statements in a slice are found only in that slice. This could be computed as the mean of the ratios of non-unique to unique statements in each slice. A high overlap might indicate very interdependent code.
Clustering	Reveals the degree to which slices are reflected in the original code layout. It could be expressed as the mean of the ratio of statements formerly adjacent to total statements in each slice. A low cluster value indicates slices intertwined like spaghetti, while a high cluster value indicates slices physically reflected in the code by statement grouping.
Parallelism	Is the number of slices which have few statements in common. Parallelism could be computed as the number of slices which have a pair wise overlap less than a certain threshold. A high degree of parallelism would suggest that assigning a processor to execute each slice in parallel could give a significant program speed-up.
Tightness	Measures the number of statements which are in every slice, expressed as a ratio over the total program length. The presence of relatively high tightness might indicate that all the slices in a subroutine really belonged together because they all shared certain activities.

Table 2. Slicing-based metrics proposed by Ott and Thuss [5]

Metric	Description
MaxCoverage	Is the length of the longest slice as a proportion of the program length
MinCoverage	Is the length of the shortest slice as a proportion of the program length

3. Inertia and maintainability

We propose the concept of *Inertia* as a means to characterise the maintainability of a system. It consists of two components, the system size and a measure of the ease or difficulty in changing the system due to its structure and code. Previous work [1, 2] confirms that over the long term systems tend to grow in size, and that as they grow they become correspondingly more difficult to maintain. This is not only because larger systems are likely to be more difficult and costly to maintain than smaller, but also because changes made to software systems over time tend to degrade its structure and makes it less maintainable unless work is performed to counteract this. To model quantitatively how easy a system will be to maintain over time, it is important to account for both changes in its size and changes in its structure. Therefore any single quantitative measure of inertia must take account of both of these dimensions.

In our existing simulation models we have used Turski's characterisation of software system growth [6] as the basis for our measure of the effect of changes in the system on the ease of making further changes to it. Turski's calculation, based purely on the physical size of systems, does not directly address the maintainability of the system. In particular, it does not account for issues of system structure and code quality.

In this work we are attempting to capture more of the phenomenon of inertia than Turski's simple abstraction. Program slicing examines and quantifies the internal linkages of the system which make maintenance of one part of a system, without consideration of the rest of it, problematical. Slicing metrics are therefore a good candidate for our purpose.

4. Applying slice-based metrics to inertia

In this section we describe how some of Weiser's [8] and Ott and Thuss' [5] slice-based metrics may be related to the effort needed to maintain a software system. Specifically, we consider the relationship of each metric to the difficulty of making changes to an existing system. In effect, we relate the metric to our notion of the 'inertia' of that system.

- **Coverage**: the existence of many short slices may indicate a system whose structure has been compromised over time by repeated cycles of changes. We conclude that lower coverage implies greater inertia, as more of the code of the system needs to be examined when changing it, i.e. an inverse correlation may be expected between coverage and inertia.
- **Overlap**: higher values of overlap mean that individual elements of code are reused in different traces through the program. Thus, when maintaining the system, if a code fragment is identified as needing change, each instance of use of that fragment will need to be located, examined and possibly replicated if the desired modification does not relate to it. Overall, a direct correlation may be expected between overlap and inertia.
- **Clustering**: lower clustering means higher inertia, because understanding and modifying less well-structured and more mutually interdependent code is likely to be more difficult. This is because the code will be more difficult to understand before changes can be designed. This will lead to greater expenditure of effort and a greater risk of errors being made in the design and implementation of changes. We therefore expect clustering to exhibit an inverse correlation with inertia.
- **Parallelism**: this may indicate that areas of functionality are well-separated in the design and the code. If this is the case, maintenance changes which respect the existing division of the problem can be made more easily. Therefore, we expect systems exhibiting high parallelism to be more easily evolvable, i.e. the relationship between parallelism and inertia is inverse.
- **Tightness**: this is related to the cohesiveness of the code. As in the case of parallelism, the benefit of more cohesive code can only be exploited if changes which have to be made to a system follow the assumptions implicit in the division of the system functions. In this case, we suggest that it is less likely that a code unit which is truly cohesive will need to be broken up due to the need for system maintenance in unexpected directions than is the case for the higher-level design decomposition measured by parallelism. Thus, there may be fewer changes needed overall if the common version can be evolved so as to continue to suit all of its uses. We suggest that code exhibiting high tightness is more likely to be easily evolvable than code with lower tightness.
- **MaxCoverage**: the higher this value, the longer the maximum path length a developer will need to understand in order to be able to appreciate the effect of any change on it and thus evolve the program safely. A high value may also reflect the existence of large blocks of structured code, which is more likely to cause the developer to need to break them up with consequent reworking of code inside a block and the design of new control structures. This metric will therefore be expected to have a direct correlation with inertia.
- **MinCoverage**: a high value for MinCoverage, reflecting a comparatively long 'shortest slice', will be subject to the same problems as those for a high value for MaxCoverage. Conversely, a low value for MinCoverage will mean that at least

some maintenance changes to the software may be localised to comparatively short traces through the code. We therefore expect MinCoverage also to be directly correlated to inertia.

In quantifying the maintainability of a system over time, it may be necessary to select, average, weight and/or total some or all of these measures on the basis of an examination of their trends.

Our conclusions concerning the relationships between these metrics should be seen in the context of Meyers' and Binkley's [3] empirical findings. Meyers and Binkley examined, *inter alia*, correlations between slicing metrics obtained for a number of open-source systems. They found strong correlations between Tightness and MinCoverage and between Tightness and Overlap, and statistically weak correlations between Tightness and Coverage, and MinCoverage and Coverage. They also concluded that Overlap was not correlated to either Coverage or MaxCoverage. They did not consider Clustering and Parallelism. With the exception of our opinion that there is an inverse relationship between Coverage and the other metrics, their results provide some practical support for our arguments.

Their results further suggest that as the size of systems grow, and as they grow older, the deterioration in structure becomes proportionally greater, which lends support to our belief that there is a relationship between trends in slicing metrics and the maintainability of systems, and that slicing metrics can therefore be used as one of the inputs to the calculation of inertia.

5. Conclusions and future work

We have shown that slice-based metrics are a promising way to measure the maintainability of software systems. We have integrated slice-based data with size data to propose inertia as a single, indirect measure of the maintainability of software systems. We expect this measure of inertia in our system dynamics models to improve the predictions of the long-term maintenance of software systems made by these models.

To answer our initial research questions:

1. Are slice-based measures a viable approach to generating data whose values and trends characterise maintainability? Although the work we present here is preliminary, our findings are promising. Slice-based measures look to be a convincing approach to characterising maintainability. Our re-interpretation of Meyers and Binkley's [3] findings suggests that these metrics will assist in measuring maintainability.

The work we present here is theoretical and we will be able to test our answer to this question more fully once we have collected empirical slicing-based metrics data and recalibrated our models. This will extend further the work already done by Meyers and Binkley [3].

2. Can maintainability data contribute to predicting the long-term maintenance of software systems? Again our preliminary results are promising. The addition of maintainability data into our system dynamics models should generate more realistic simulations. This means that our work simulating the long-term maintenance of software systems will be capable of being applied with greater confidence to the investigation of the impact of process change on long-term software maintenance.

References

[1] Chatters BW, Lehman MM, Ramil JF, Wernick P, "Modelling A Software Evolution Process", *Software Process: Improvement and Practice*, **5**, 2000, pp.91–102.

[2] Lehman MM, Perry DE, Ramil JF, Turski WM and Wernick PD, "Metrics and Laws of Software Evolution - The Nineties View", *Proc. Metrics '97* Albuquerque, NM, 5–7 Nov, 1997.

[3] Meyers TM and Binkley D, "Slice-Based Cohesion Metrics and Software Intervention", *Proc. IEEE 11th Working Conference on Reverse Engineering,* Delft, Netherlands 9–12 Nov 2004.

[4] Ott L and Thuss J, "The relationship between slices and module cohesion.", *Proc. ICSE 1989*, Pittsburgh, Pennsylvania, 1989, pp.198–204.

[5] Ott L and Thuss J, "Slice based metrics for estimating cohesion", *Proc. First International Software Metrics Symposium*, Baltimore, MD, May 1993, pp.71–81.

[6] Turski WL, "The Reference Model for Smooth Growth of Software Systems Revisited", *IEEE Trans. Software Engineering*, **28** (8), 2002, pp.814 – 815.

[7] Wernick P and Hall T, "The Impact of Using Pair Programming on System Evolution: a Simulation-Based Study", *Proc. ICSM 2004*, Chicago, IL, Sept. 11–14, 2004.

[8] Weiser M, "Program slicing", *Proc. ICSE 1981*, San Diego, California, Mar. 9–12 1981, pp.439–449.

[9] Weiser M, "Programmers use slices when debugging", *Comm. ACM*, **25** (7), 1982, pp.446-452.

Model Synchronization through Pattern-Based Association Grammars*

Igor Ivkovic and Kostas Kontogiannis
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, ON N2L3G1 Canada
{iivkovic, kostas}@swen.uwaterloo.ca

Abstract

Changes made to evolving software systems are usually applied to models that pertain to different levels of abstraction. Transformations made at one model (e.g., source code) must be correctly interpreted and applied to all other affected models (e.g., design, architecture) in order to minimize the drift among system artifacts. This position paper focuses on interpreting model synchronization as a problem of automated language translation. In this approach applicable MOF-compliant domain models are formally defined and converted to graph grammars, called domain model grammars. A pattern-based association grammar, derived from NLP theory and grammar-based MT, is used to translate models instantiated from different domains. Source and target productions are associated by matching MOF attributes, and conflicts in pattern selections are resolved through constraint predicates.

1. Introduction

Stakeholders in an evolving software system directly or indirectly initiate changes on various software artifacts at different levels of abstraction and technical detail. For every defined change that is applied, all affected models must be updated in a systematic fashion. The most complex facet of this process is the propagation of change across models used in different stages of the software lifecycle since the models differ greatly in expressiveness levels and model semantics. This problem, discussed as a framework for *model synchronization through traceability* - ***mSynTra*** [4, 5, 6], is based on the Model-Driven Software Evolution (MDSE) and Model-Driven Architecture (MDA) [7, 11] paradigms. Within mSynTra, software changes are made on models at a particular level of abstraction within the context of an iterative and incremental lifecycle such as the Rational Unified Process (RUP) [3]. Synchronization of two models changed due to evolution is done by tracing a sequence of transformations applied to one model, and translating them into a sequence that is applied to other affected models.

This paper focuses on adding translation capability to the mSynTra framework by interpreting the problem of model translation in terms of language translation, and applying established algorithms and theory from the area of NLP to a more structured domain of software. The goal is to represent models as sentences that comply to corresponding grammars, and then view model translation as grammar-based language translation. The first step is to represent domain models as unique sets of tuples for domain types, relations, connectors, and attributes, which are then interpreted as graph grammars, called domain model grammars. Models instantiated from the domain models can then be viewed as sentences in the grammar-generated languages. Pattern-based association grammars and rules are used to establish relations between the source and target domain model grammars. Conflicts in pattern selection are resolved through predicates, which can be used to express additional constraints.

2. mSynTra Framework Overview

The mSynTra framework [5] is based on our view of software artifact related MOF-compliant models [12] as directed, labelled, attributed graphs. All graph properties are expressed in terms of labels and (attribute, value) pairs associated with respective nodes and edges. Models are represented using MOF-compliant metamodels and are instantiated from their respective domain models, which represent domain types and relations specified in a chosen metamodel notation such as UML [13]. The model transformations applied to the concrete models are interpreted as basic graph transformations (*i.e.*, insertions, modifications, and deletions of model elements and their attributes). Each trans-

[1] This work is funded in part through the IBM Corp. Scholars Program and the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

formation is applied on properties and predicates that are defined at the domain model level, and it can therefore be mapped or traced to its domain model or to the corresponding metamodel. Conclusively, applied model transformations can be viewed in terms of their atomic operations (*i.e.*, graph transformations) and can be interpreted as combinations of these basic elements. The synchronization activities can be segmented into two categories: (1) transformations and (2) translations. Transformations are performed by a transformer entity and are related to applying changes performed on one model within the same domain (*e.g.*, applying individual transformations, tracing transformation sequences). Translations are performed by a translator entity and are related to applying and propagating changes of one model from one domain model into a new model in a different domain (*e.g.*, establishing model dependencies, interpreting changes from one domain model to another).

3. Domain Models as Graph Grammars

In this section, we describe the first part of our approach to model synchronization as a problem of language translation, namely, the process of representing domain models and metamodels as corresponding grammars. The idea of representing software models as graph grammars was previously described by Metayer [9] while Alanen and Porres [2] have derived a method for interpreting MOF metamodels directly as Extended Backus-Naur Form (EBNF) grammars.

Our conceptual view of MOF-compliant models is that of graphs, so we interpret domain models specifically as graph grammars. Using previously defined graph metamodel for synchronization (GMS), we view domain models as collections of attributed nodes (domain types) and attributed and directed edges (ordered domain relations), which represent types for instantiated concrete models. The domain model elements are then viewed as nonterminals, and the concrete model elements are viewed as terminals in the domain model grammar (DMG).

Definition 1 *(Domain Model Grammar) A domain model grammar (DMG) for a domain model DM := (DT, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values) is a tuple (NT, T, P, AX), where a set of nonterminals NT := $\{nt \mid nt \in (T_{Names} \cup R_{Names} \cup C_{Names} \cup A_{Names})\}$, a set of terminals T := $\{t \mid t \in Values\}$, a finite set of production rules P := $\{(LHS, RHS) \mid$ where $LHS \in NT$, $RHS \in (NT \cup T)^*\}$ inferred from DT, DR, DC, and DA, and AX is the axiom that represents the origin for the derivation.*

4. Representing Relation Types

Using DM2DMG algorithm on a particular section of the UML metamodel as input [13], a grammar required to represent the UML relation types is as follows:

Step 1-2 :
NT_T := {Classifier}, NT_R := {Association, Generalization}, NT_C := {AssociationEnd, GeneralizationEnd}, NT_A := {Name, Constraint, Aggregation, IsNavigable, Multiplicity, Visibility}, NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid UML element names}, AX := M.

Step 3. :
p_1 : M → Classifier | Classifier M | Association | Association M | Generalization | Generalization M

Step 4. :
p_2 : Classifier → Name

Step 5. :
p_3 : Association → Name Name AssociationEnd AssociationEnd

p_4 : Generalization → Name null GeneralizationEnd GeneralizationEnd

Step 6. :
p_5 : AssociationEnd → Name Aggregation IsNavigable Multiplicity Visibility Classifier

p_6 : GeneralizationEnd → Constraint Classifier

Step 7. :
p_7 : Name → alphabet of valid element names (values)

p_8 : Constraint → alphabet of valid constraints

p_9 : Aggregation → none | shared | composite

p_{10} : IsNavigable → true | false

p_{11} : Multiplicity → none | [nonnegative integer] [nonnegative integer] | [nonnegative integer] *

p_{12} : Visibility → public | protected | private | package

Step 8 :
P := $\{p_1, p_2, \ldots p_{12}\}$ and DMG := {NT, T, P, AX}.

Step 9-10 :
Manually confirmed that DMG is unambiguous (*e.g.*, by converting to the Chomsky normal form (CNF) [8]) so output DMG and terminate.

Figure 1 illustrates a derivation tree based on the described grammar for the "Association" UML relation.

5. Pattern-Based Association Grammars

The synchronization of two heterogeneous models defined at different levels of abstraction is interpreted as the translation between two domain model grammars (DMGs) to which the two models conform. Based on the theory of grammar association [14], the translation methodology then operates on the following:

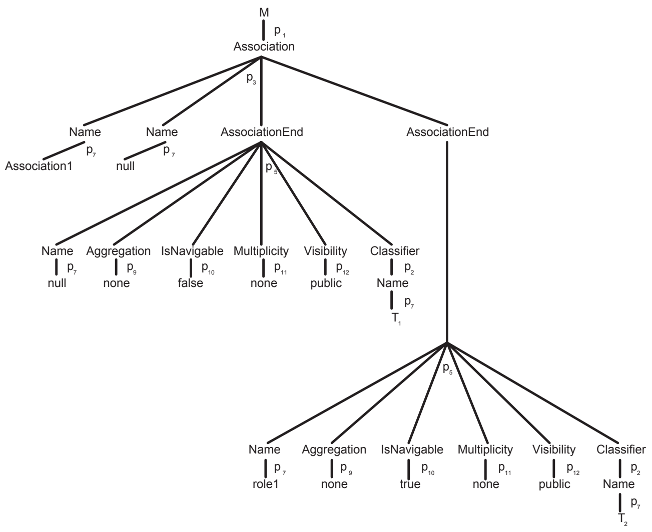


Figure 1. A Derivation Tree for R$_1$

- A source DMG, modelling the language of the source models;
- A target DMG, modelling the language of the target models; and
- An association model, describing the associations between the rules of the input and output grammars.

Previously, Akehurst [1] addressed the problem of model translation by utilizing a combination of UML and OCL [16] to specify transformation relations between two object-oriented models. A technique was also proposed by Milicev [10] to utilize extended UML object diagrams to specify translation between source and target metamodels.

We make use of pattern-based context-free grammar (PCFG) [15] to formalize the association model and enable model translation. Each PCFG consists of a set of translation patterns that are expressed as a pairing of CFG rules, the source rule on the LHS and the target rule on the RHS, along with zero or more syntactic (head) and link constraints. Each of the constraints is expressed in terms of nonterminal symbols that represent different lexical elements from the domain of natural languages, examples of which include noun phrase (NP), verb phrase (VP), *etc*. A translation pattern can also be associated with a vector of binary features to represent additional syntactic and semantic constraints.

Definition 2 *(Pattern-Based Assocation Grammar) A pattern-based association grammar (PAG) for source and target domain model grammars, DMG$_S$:= (NT$_S$, T$_S$, P$_S$, AX$_S$) and DMG$_T$:= (NT$_T$, T$_T$, P$_T$, AX$_T$) respectively, is a tuple (NT, T, AP, X, S_{Heads}, Predicates), where a set of nonterminals NT := $\{nt \mid nt \in (NT_S \cup NT_T)\}$, a set of terminals T := $\{t \mid t \in (T_S \cup T_T)\}$, a finite set of predicated association rules AP := $\{(s_H,\ r_S,\ r_T\ \{p_i\}) \mid$ a semantic head $s_H \in S_{Heads}$, a source production rule $r_S \in P_S$, a target production rule $r_T \in P_T$, and $p_i \in$ Predicates$\}$, a starting symbol for derivation AX := AX$_S$, a set of semantic heads (attribute-value constraints) S_{Heads} := $\{\{(a_i,\ v_j,$ optional$)\} \mid a_i \in$ a finite set of valid attributes and $v_i \in$ a finite set of valid values defined at the MOF or metamodel level, and optional as a Boolean indicating wether the satisfaction of an attribute constraint is optional$\}$, and a finite set of constraint Predicates expressed in a suitable language (e.g., OCL expression).*

5.1. Pattern-Based Translation Process

To translate an input model M $\in$ L(DMG$_M$) into a model G $\in$ L(DMG$_G$) using an association grammar PAG$_{M,G}$, the following steps apply:

1. Parse M using production rules from DMG$_M$ and create a leftmost derivation tree T$_M$ while labelling the edges of T$_M$ with the identifiers for used source productions r$_S$.
2. Parse T$_M$ using a depth-first tree-parsing algorithm:
   (a) At each node identify a source production r$_i$ used for deriving that node and look for a matching association production ap = (s$_H$, r$_S$, r$_T$, {predicates}) from the PAG such that r$_i$ = r$_S$.
   (b) If more than one match is found, select the best match by applying predicates to L(r$_T$) for each match or by using a particular translation goal (*e.g.*, minimizing the number of elements in G).
   (c) If no match is found, recursively attempt to match the rule used to derive the parent of the current node and use predicates or higher-level translation goals to resolve conflicts; if the root node is reached with no match, declare the current node as unmatchable and continue.
   (d) Once the matching target rule is found, use it to synchronously derive the target tree T$_G$.
3. Use the resulting derivation tree T$_G$ to derive G.

5.2. Determinism and Algorithmic Complexity

The determinism of the resulting derivations is addressed through (1) the adjustable level of semantic detail expressed as a chosen number of attribute-value pairs as defined at the MOF or at the metamodel level, (2) predicates associated with each rule that add additional constraints in conflict resolution, and (3) global translation rules that impose global constraints for individual rule associations.

Each PAG is defined as a PCFG so its asymptotic complexity corresponds to asymptotic limits as defined for PCFG in the area of NLP. Therefore, the translation complexity is based on the complexity for choosing the top pattern candidates, O(|T|Kn^3), selecting the suitable patterns and relating them to the target translation, O(|T|Kn^4), and constructing the target derivations based on the m candidate patterns, O(Kn^2m), where T is the PCFG, K are distinct nonterminals in T, and n is the size of the input string [15].

6. Conclusions and Future Research

This paper presents a framework for interpreting the problem of model synchronization as a language translation problem. We first discussed the representation of a domain model by a graph grammar. In this context, the corresponding models are considered as "sentences" generated by a domain graph grammar that need be translated from one language to another (*i.e.*, from one domain model to another). Second, we presented the creation of a corresponding pattern-based association grammar, based on attributed associations of production rules from the source and target domain model grammars, and discussed the application of the association grammar in the process of translation. Finally, we have evaluated the approach by applying it to a case study of synchronizing business process models with enacting Java source code.

In future research, we aim to extend the capabilities of this approach by applying it to additional case studies that relate to different stages of the software development lifecycle. Specifically, we intend to investigate the process of iterative and incremental translation of extended BPM models to source code though intermediate, stereotyped and annotated UML models.

7. Acknowledgments

This work is performed in collaboration with the IBM Canada Ltd. Laboratory, Center for Advanced Studies (CAS) in Toronto.

References

[1] D. H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent at Canterbury, Dec 2000.

[2] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. TUCS Technical Report No 606, Turku Center for Computer Science, Åbo Akademi University, Turku, Finland, 2003.

[3] IBM. Rational unified process (rup). Online by IBM Corporation, 2004. http://www.ibm.com/software/awdtools/rup/.

[4] I. Ivkovic and K. Kontogiannis. Model synchronization as a problem of maximizing model dependencies. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 222–223, Vancouver, BC, Oct 2004.

[5] I. Ivkovic and K. Kontogiannis. Tracing evolution changes through model synchronization. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 252–261, Chicago, IL, Sep 2004.

[6] I. Ivkovic and K. Kontogiannis. Using formal concept analysis to estalish model dependencies. In *Proceedings of the IEEE International Conference on Information Technology Coding and Computing*, pages 365–372, Las Vegas, NV, Apr 2005.

[7] A. Kleppe, J. Warmer, and W. Bast. *The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[8] J. C. Martin. *Introduction to Languages and the Theory of Computation*. WCB/McGraw-Hill, 1997.

[9] D. L. Métayer. Software architecture styles as graph grammars. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 15–23, San Francisco, CA, Oct 1996.

[10] D. Milicev. Automatic model transformations using extended uml object diagrams in modeling environments. *IEEE Transaction on Software Engineering*, 28(4), Apr 2002.

[11] OMG. Model driven architecture - a technical perspective. Object Management Group's (OMG's) Architecture Board ORMSC Document ORMSC/01-07-01, Object Management Group, Jul 2001.

[12] OMG. Meta object facility (mof) specification version 1.4. Technical report, Object Management Group (OMG), Apr 2002. http://www.omg.org/docs/formal/02-04-03.pdf.

[13] OMG. Unified modelling language (uml) specification. Technical report, Object Management Group, Mar 2003. http://www.omg.org/docs/formal/03-03-01.pdf.

[14] F. Prat. Machine translation with grammar association: Some improvements and the loco_c model. In *Proceedings of the Workshop on Data-driven Machine Translation at 39th Annual Meeting of the Association for Computational Linguistics*, Toulouse, France, Jul 2001.

[15] K. Takeda. Pattern-based context-free grammars for machine translation. In *Proceedings of the 34th conference on Association for Computational Linguistics*, pages 144–151, Santa Cruz, CA, Jun 1996.

[16] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

Supporting Continuous Evolution of Software Systems with Transformation Maintenance

László Lengyel, Tihamér Levendovszky and Hassan Charaf
Budapest University of Technology and Economics,
H-1111 Budapest, Goldmann György tér 3.
{lengyel, tihamer, hassan}@aut.bme.hu

Abstract

Software maintenance and evolution is a considerably researched area while taking into account its cost effects. Specifying systems in a higher abstraction level, it helps understanding, developing and maintaining the software. A higher abstraction level can be achieved by software models and their model transformation. Model compilers provide a solution for automated source code generation from software models and mechanisms for software maintenance. This paper introduces a metamodel-based model transformation approach for the continuous software evolution support, discusses the configurability of the model transformation with the help of pre- and postconditions (OCL constraints) propagated to transformation steps and introduces the concept of aspect-oriented constraints.

1. Introduction

In software development processes new requirements appear frequently, when the design phase is already closed e.g. in the middle of the implementation or later. Moreover, it is a common procedure in development that an initial version of an application is developed in the first step, and afterwards the new features or modifications of the old ones are added step by step to the already existing implementation. These methods of the software development processes require the efficient support of software evolution and maintenance of the implementations.

Software maintenance activities are mainly accomplished manually at present, yet they are the largest software engineering expense. Considerable productivity and quality enhancements are possible with a new generation of automated tools called model compilers. This approach accelerates the development and maintenance process and eliminates the need of the manual work on the source code.

A model compiler is a tool that automatically converts a set of modeling artifacts into an equivalent artifact [1], which can be model or source code as well.

Model transformation means converting an input model available at the beginning of the transformation process to an output model. OMG's Model Driven Architecture [2] sets out a more restrictive definition: the output model should describe the same system as the input model. But our approach (VMTS [3] [4]) has been designed to be able to specify more general transformations than the property preserving ones. Model compilers can support certain model element properties to guarantee, preserve or validate them during the code generation, and the presented approach is a practical application of these mechanisms [5].

Model Integrated Computing (MIC) [6] [7] is a model-based approach to software development, facilitating the synthesis of application programs from models created using customized, domain-specific program synthesis environments.

This work presents the VMTS approach to artifact and source code maintenance, shows that the metamodel-based model transformation is an efficient method for software evolution support. VMTS uses graph rewriting as underlying method for model transformation (source code generation); this paper discusses the power of constraints contained by the rewriting rules during the model compilation process, and illustrates that with the help of the constraints it is possible to specify fully the source code generation. It also means that the same rewriting rules with other constraints generate different implementation. We introduce an aspect-oriented method which facilitates (i) the reuse of the rewriting rules and constraints and (ii) the modularization of crosscutting constraints. Our approach provides the possibilities to define constraints separately and to specify their propagation to rewriting rules.

2. Backgrounds and Related Work

MIC focuses on models, supports the flexible creation of modeling environments, and helps following the changes of the models. At the same time it facilitates code generation and provides tool support for turning the created models into code artifacts. Metamodeling environments and model interpreters together form the tool support for MIC.

Our metamodel-based model transformation system and constraint validation method presented later benefit from the results of the mathematical background of formal languages, graph rewriting and research related to the metamodel-based software model transformation. It also incorporates several ideas from other existing environments (the PROGRES [8] and GReAT framework [9]).

The Object Constraint Language [10] is a formal language for analysis and design of software systems. It is a subset of the industry standard Unified Modeling Language [11] that allows software developers to write constraints and queries over object models.

Graph rewriting [12] is a powerful tool for graph transformations with strong mathematical background. The atoms of graph transformation are rewriting rules, each rewriting rule consists of a left hand side graph (LHS) and right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of the LHS in the graph the rule being applied to (host graph), and replacing this subgraph with RHS. Replacing means removing elements which are in the LHS but not in the RHS, and gluing elements which are in the RHS but not in the LHS.

Models can be considered special graphs that simply contain nodes and edges between them. This mathematical background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models using graph rewriting [4] [13]. Previous work [4] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements. It means that an instantiation of LHS must be found in the graph (host graph) to which the rule being applied instead of the LHS-isomorphic subgraph. Hence the LHS and RHS graphs are the metamodels of the graphs which we search and replace in the host graph.

In [14] an aspect oriented approach is introduced for software model containing constraints where the dominant decomposition is based upon the functional hierarchy of a physical system. This approach provides a separate module for specifying constraints and their propagation. To provide the weaver with the necessary information to perform the propagation, a new type of aspect is used: the strategy aspect. Strategy aspect provides a hook that the weaver may call in order to process the node specific constraint propagations.

3. Visual Modeling and Transformation System

The Visual Modeling and Transformation System (VMTS) [3] [4] is an implemented n-layer multipurpose modeling and metamodel-based model transformation system. Using this environment, it enables to edit metamodels and models according to their metamodels and transform models using graph rewriting [4]. Furthermore, the tool facilitates to check constraints specified in the metamodel during the metamodel instantiation, and the rewriting rule constraints during the graph transformation process.

VMTS supports both the Traversing Model Processors (TMP) and Visual Model Processors (VMP).

The simplest method to transform models is to traverse them using a specific programming language and changing the appropriate parts of the input models or producing an output model. TMPs offering this approach usually use the following basic graph operations: node creation, node deletion, edge creation, edge deletion and label modification.

VMPs do not replace TMPs, instead, they provide a visual alternative way of model transformation. In VMTS VMPs use graph rewriting as the transformation technique. The graph rewriting production rule firing has already been introduced in Section 2.

4. Rewriting Rule-Based Software Maintenance

The fact that VMTS is able to work as a model compiler means that it facilitates to generate the whole application based on the software models. Obviously, it is crucial that the models contain all necessary information which makes possible to generate all the details of the software. VMTS uses UML models as input; in most cases we need several classes, statecharts, use cases and sequence diagrams to specify an application in details.

It is required that in the process of software development the software models correspond to the implementation, and vice versa. In general if a software model is modified programmers based on the modifications update the source code. This process requires not only the exact model modification but accurate source code update as well. It means that the software maintenance will be longer and more expensive. Using model compilers, after modifying the software models, we can generate the whole application again, which means that with the help of this approach, we do not need to manually work on the source code.

As it is mentioned in previous section, one of the reasons why we have to maintain the source code of the implementation is when (i) the software models are modified. In this case using the VMTS approach we simply have to generate again the implementation from the new software models. There are two other cases when we have to update the implementation: (ii) the metamodel of the software models is modified, or (iii) we would like to construct the software models from another point of view and this is why we want to develop a new application based on the same models. Because of the new viewpoint we would like the operation of the newly created application to be partly different. In other words in this case we need a new transformation process, which contains updated and/or new transformation steps.

The VMTS approach uses rewriting rules to realize a model compiler. If the metamodel of the software models is modified, the developers have to update the software models as well. Metamodel modification means the following operations: the properties of a type or a connection between the types are modified, new type is added to or deleted from the metamodel, and a connection is added to or deleted from the metamodel. Based on these modifications, the developers maintain the software models: modify the changed properties of the nodes and edges, delete the instance nodes of the removed types and if it is necessary add new nodes and edges to the models. The modifications of the metamodel also affect rewriting rules, because the rules are built from metamodel elements. After updating the rewriting rules – which also means type and connection modification, deletion and creation – we have to simply generate again the whole application with the new transformation steps.

In the third case, when we would like to generate different application based on the same models, we also have to modify the rewriting rules based on our new viewpoint and generate again the application.

Constraints (pre- and postconditions) makes possible to specify the rewriting rules precisely enough to recognize and to signal e.g. if the input models are incomplete or contain some contradiction or defective parts. With the help of these constraints we achieve precise and consistent models and transformation steps. In VMTS the principle of the constraint validation is the relation between the pre- and postconditions and the OCL constraints assigned to the rewriting rules.

The base of the rewriting-rule-based software maintenance is the graph rewriting process introduced in Section 2 (matching, removing and gluing). The matching process selects those parts of the input models from which the rewriting rule generates the source code. In the case of diagrammatic languages, such as the UML, the exclusive topological matching is found to be not enough. To define precisely the transformation steps beyond the topology of the visual models additional constraints must be specified which ensures the correctness of the attributes, or other properties to be enforced. Dealing with OCL constraints provides a solution for the unsolved issues, because topological and attribute transformation methods cannot perform and express the problems, which can be addressed by constraint validation. The use of OCL as a constraint and query language in modeling and model transformation is essential.

We have found that often it is not necessary to modify the topology of the rewriting rules, but the constraints assigned to the rewriting rules; therefore it would be beneficial to create and manage rewriting rules separately from the constraints and make it possible to propagate the required constraints to an optional rewriting rule and to use the so-called *Weaving Configuration* during the transformation process. In this case it would be unnecessary to recreate and store the same rewriting rule several times with other assigned constraints. On the other hand, often, the same constraint is repetitiously applied in many different places in a transformation. It would be useful to describe a common constraint in a modular manner and designate the places where it is to be applied. These are the reasons why we have worked out the concept of the Aspect-Oriented Constraints.

5. Aspect-Oriented Constraints

We need a mechanism to separate the constraints from the pattern rule nodes (PRNs) and a weaver method, which facilitates the propagation (linking) of constraints to PRNs. The VMTS Global Constraint Weaver is passed a transformation with optional number of transformation steps and a constraint list and it links the constraints to transformation steps containing PRNs [15].

This method means that VMTS manages constraints using aspect-oriented techniques: constraints are specified and stored independently of any transformation step or PRN and they are linked to the PRNs by the Global Constraint Weaver [15].

The weaving algorithm based on the context of the constraint selects the PRNs the constraint has to be linked to. Furthermore, the algorithm also has to take into account the transformation steps which can modify the properties for that the constraint has restrictions. In other words examining the transformation steps, the algorithm decides if it is necessary to assign the constraint to each step or it is sufficient to assign only to the first as precondition and to last step as postcondition. If an intermediate state modifies one of the properties contained by the constraint, then the weaving algorithm assigns the constraint to this intermediate state to prevent that the not satisfied condition does not turn out until the end of the transformation.

It is unnecessary that the transformation has any knowledge about the constraints, nor the modeler who creates the rewriting rules. Rewriting rules can be executed without constraints as well, but in that case the matching can be accomplished only by the topological information.

To summarize the main idea of the AO Constraints is that we can separately create the constraints and the rewriting rules, and with the help of a weaver we can propagate constraints to the rewriting rules containing PRNs [15].

6. Conclusions

In this work the VMTS approach to rewriting rule-based software maintenance is presented and the concept of aspect-oriented constraint management in metamodel-based model compilers is introduced. It is discussed that using our approach, it is possible to maintain the implementations when (i) the software models are modified, (ii) the metamodel of the software models is modified, or (iii) if it is required to consider the software plans from an other point of view, and to implement a different application from the same software models.

The presented concepts have been applied to mobile devices running Symbian operating system, where the resource constraints made the visual model-based evolution support more useful towards an iterative incremental development cycle [16].

This approach generates the whole application again, it would be beneficial if it generated source code only from the new and the modified parts of the models, this problem is the subject of future research.

Acknowledgement

The fund of “Mobile2004 Consortium” has supported, in part, the activities described in this paper.

References

[1] Butts K, Bostic D, Chutinan A, Cook J, Milam B, Wang Y, “Usage scenarios for an Automated Model Compiler”, *EMSOFT 2001*, pp 66–79

[2] MDA Guide Version 1.0.1, OMG, doc. number: omg/2003-06-01, 12th June 2003 www.omg.org/docs/omg/03-06-01.pdf

[3] Visual Modeling and Transformation System Web Site, http://avalon.aut.bme.hu/~tihamer/research/vmts/

[4] Levendovszky T, Lengyel L, Mezei G, Charaf H, “A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS”, *International Workshop on Graph-Based Tools (GraBaTs) Electronic Notes in Theoretical Computer Science*, Rome, 2004

[5] Lengyel L, Levendovszky T, Charaf H, Implementing an OCL Compiler for .NET, Journal of .NET Technologies, Volume 3, Number 1-3, 2005, ISSN 1801-2108, pp. 121-130

[6] Sztipanovits J, Karsai G, “Model-Integrated Computing” *IEEE Computer*, pp. 110-112, April, 1997.

[7] Sztipanovits J, Karsai G, “Generative Programming for Embedded Systems”, *LNCS 2487*, pp. 32-49, 2002

[8] PROGRES system can be downloaded from http://mozart.informatik.rwth-zaachen.de/research/projects/progres/main.html

[9] Karsai G, Agrawal A, Shi F, Sprinkle J, “On the Use of Graph Transformation in the Formal Specification of Model Interpreters”, *Journal of Universal Computer Science*, Special issue on Formal Specification of CBS, 2003

[10] Object Constraint Language (OCL), www.omg.org

[11] UML 2.0 Specifications, http://www.omg.org/uml/

[12] G. Rozenberg (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol.1 World Scientific, Singapore, 1997.

[13] D. Varró and A. Pataricza, VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML, *Journal of Software and Systems Modeling,* 2003

[14] Jeff Gray, Ted Bapty, Sandeep Neema, “Aspectifying Constraints in Model-Integrated Computing”, *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Minneapolis, MN, October 2000

[15] Lengyel L, Levendovszky T, Charaf H, Weaving Crosscutting Constraints in Metamodel-Based Transformation Rules, *8th International Conference on Information Systems Implementation and Modeling, ISIM '05*, April 19-20, 2005, Czech Republic, pp. 119-126

[16] Aczél K, Charaf H, “Automatic User Interface Code Generation in Symbian”, *MicroCAD*, Miskolc, 2005

Analyzing the Reuse Potential of Migrating Legacy Components to a Service-Oriented Architecture

Grace Lewis, Edwin Morris, Liam O'Brien, Dennis Smith
Software Engineering Institute
4500 Fifth Avenue,
Pittsburgh, PA 15213
{glewis, ejm, lob, dbs}@sei.cmu.edu

ABSTRACT

An effective way of leveraging the value of legacy systems is to expose their functionality, or subsets of it, as services. In the business world, this has become a very popular approach because it allows systems to remain largely unchanged, while exposing functionality to a larger number of clients through well-defined service interfaces. The U.S. Department of Defense (DoD) is also adopting this approach by defining service-oriented architectures (SOAs) that include a set of infrastructure common services on which organizations can build additional domain services or applications. When legacy systems or components are to be used as the foundation for these services, there needs to be an analysis of how to convert the functionality in these systems into services. This analysis should consider the specific interactions that will be required by the SOA and any changes that need to be made to the legacy components. The SEI has recently helped an organization evaluate the potential for converting components of an existing system into services that would run in a new and tightly constrained DoD SOA environment. This paper describes the process that was used and outlines several issues that need to be addressed in making similar migrations.

1. Introduction

With the advent of universal Internet availability, many organizations have leveraged the value of their legacy systems by exposing all or parts of it as *services*. A service is a coarse-grained, discoverable, and self-contained software entity that interacts with applications and other services through a loosely coupled, often asynchronous, message-based communication model [2]. A collection of services with well-defined interfaces and shared communications model is called a service-oriented architecture (SOA). A system or application is designed and implemented as a set of interactions among these services.

The characteristics of SOAs (e.g., loose coupling, published interfaces, standard communication model) offer the promise of enabling existing legacy systems to expose their functionality, presumably without making significant changes to the legacy systems[4]. However, constructing services from existing systems in order to obtain the benefits of an SOA is neither easy nor automatic. In fact, such a migration can represent a complex engineering task, particularly when the services are expected to execute within a tightly constrained environment.

SOA migration tasks can be considered from a number of perspectives including that of the end client or user of the services, the SOA architect, or the service provider. This paper focuses on the service provider.

2. Creation of Services From Legacy Components

Enabling a legacy system to interact within a service-oriented architecture, such as a Web services architecture, is sometimes relatively straightforward—this is a primary attraction to the approach for many businesses. However, characteristics of legacy systems, such as age, language, and architecture, as well as of the target SOA can complicate the task. An analysis needs to be performed to consider:

1. requirements from potential service users. It is important to know what applications would use the services and how they would be used. For example, what is the information expected to be exchanged? In what format?
2. Technical characteristics of the target environment, such as bindings, messaging technologies, communication protocols, service description languages, and service discovery mechanisms.
3. The architecture of the legacy system., including dependencies on commercial products or specific operating systems, or poor separation of concerns.
4. The effort involved in writing the service interface
5. The effort involved in the translation of data types.
6. The effort required to describe the services including information about qualities of service, such as performance, reliability, and security; or service level agreements (SLAs) .
7. The effort involved in writing service initialization code and operational procedures.
8. Estimates of cost, difficulty, and risk.

To gather this information and identify the risks for the migration effort in a systematic way, we have developed the Service-Oriented Migration and Reuse Technique (SMART). SMART is based on the OAR [1]method for evaluating the reuse potential of legacy components, but customized to reflect the migration of components to services. Its activities are:

1. Establish stakeholder context
2. Describe existing capabilities
3. Describe the future service-based state
4. Analyze the gap between service-based state and existing capabilities
5. Develop strategy for service migration

These five activities are briefly outlined below.

Establish Stakeholder Context
In order to establish the context in which the migration to services will take place, SMART first identifies the stakeholders, including the current end users of the legacy systems, the potential end users of the migrated service operating within the SOA, and the owners of the legacy systems. The activity identifies who knows most about the legacy system, what it currently does, and what it should do as a service or set of services.

Describe Existing Capabilities
The goal of the second activity is to obtain descriptive data about the legacy components. Basic data solicited includes the name, function, size, language, operating platform, and age of the legacy components. Technical personnel are questioned about the architecture, design paradigms, code complexity, level of documentation, module coupling, interfaces for systems and users, and dependencies on other components and commercial products.

Historical cost data for development and maintenance tasks is collected to support effort and cost estimates.

Describe the Future Service-Based State
The two goals of the third activity are to:

∞ Gather evidence about potential services that can be created from the legacy components
∞ Gather sufficient detail about the target SOA to support decisions about what services may be appropriate and how they will interact with the architecture

Initial information about potential services often comes via conversations about the function(s) of the legacy system during the second activity. However, the information gathered often must be tempered by data from users, corporate architects, domain groups, communities of interest, and reference models that address service definition. In some cases, these groups and models will define the entire set of services that support the organization's goals, and into which any potential services built from the legacy components must fit.

Analyze the Gap
The goal of the fourth activity is to identify the gap between the existing state and the future state and determine the level of effort needed to convert the legacy components into services. This analysis may also suggest potential tradeoffs between the target architecture and the legacy components.

The tasks of this activity include:

∞ Develop an analysis strategy for legacy components that are being considered for migration.

∞ Analyze the legacy components to determine the types of changes that need to be made to enable migration. SMART uses three sources of information to support the analysis activity. The issues, problems, and other concerns that were noted as the team completed the previous, discovery-oriented steps form one source of information. A second source of information is provided by a Service Migration Inventory (SMI) that distills the many desired traits of services executing within SOAs into a set of topics. The team uses the SMI to assure broad coverage and consistent analysis of difficulty, risk, and cost issues. A third, optional source of information involves the use of code analysis and architecture reconstruction tools to analyze the existing source code for legacy components.

Develop Strategy for Service Migration
A key feature of SMART involves building cost projections for each migration option still under consideration. This is accomplished by considering organizational characteristics, difficulty and risk associated with various migration options, and applying historical productivity numbers where possible.

3. Pilot Application of the Process

An early version of SMART was applied in a recent pilot analysis of the potential for migrating a set of legacy components from a DoD command and control (C2) system to an SOA.

Establish Stakeholder Context

We initially met with the government owners of the system and the contractors who had developed the system. At this meeting we were given an overview of the set of systems, the history of the systems, the migration plans, and the drivers for the migration. We were given a brief orientation to the SOA and were also provided with system documentation.

The owners of the systems recognized that if a selected set of components from their C2 system are converted to application domain services within a specific target

SOA, they may have applicability for a broad variety of purposes. Our role was to perform a preliminary evaluation of the feasibility of converting a set of their components to application domain services within a SOA.

Describe Existing Capabilities

The pilot C2 system has two parts: 1) a mission planning system and 2) a mission execution system that adds situational awareness to the planning capability. These two systems were initially developed as part of a product line. Both rely on a set of core components for the data model, data analysis, and visualization.

Given the information about the target SOA, we met with the contractor and representatives of the government to focus on a limited number of legacy components and to select criteria for further screening. We focused on seven potential services that the government team had previously identified as part of its initial analysis of ADS requirements. These seven potential services contained 29 classes.

The current system, written in C++ on a Windows operating system, had a total of about 800,000 lines of code and 2500 classes. In addition, the system had dependencies on a commercial database and a second product for visualizing, creating, and managing maps. Both commercial products have only Windows versions.

The 29 classes that we selected enabled us to focus on potentials for high payoff. In conjunction with the team, we developed criteria for screening the potential reusable components. These criteria included:

- Size
- Complexity
- Level of documentation
- Coupling
- Cohesion
- Number of base classes
- Programming standards compliance
- Black box vs. white box suitability
- Scale of changes required
- Commercial mapping software dependency
- Microsoft dependency
- Support software required

These criteria formed the basis for the more detailed analysis discussed below.

Describe the Future Service-Based State

The system owner had done a preliminary identification of potential services that could be built from components of the legacy system. This analysis was derived from high level requirements for applications that were being targeted as users of services to be provided by the SOA. The system owner had matched legacy functionality to these high level requirements and provided some initial estimates of the contents of the potential services.

We investigated the target SOA through an analysis of available documentation and through a meeting with the developers. Because the SOA was still under development, the specifications for how to deploy and write services were still unclear.

Analyze the Gap

Given the known and projected constraints of the target SOA, we performed three different types of analyses: 1) an analysis of the changes to the legacy components that would be necessary for migration to the SOA, 2) an informal evaluation of code quality, 3) an architecture reconstruction to obtain a better understanding of the set of undocumented dependencies. The results of these analyses allowed us to define a service migration strategy based on the risks due to the unknown future state of the target SOA..

Analysis of Required Changes

We initially met with the contractor to get an understanding of the required changes, as well as estimates of the level of difficulty and the risks of making the changes. The contractor provided estimates for converting the components into services, based on a set of simplifying assumptions on the actual make-up of the target SOA and the final set of user requirements. These estimates initially suggested that the level of difficulty of making these changes would be low to medium, and the risk would be low because of their familiarity with the systems.

However, we found that the tools in use on the project only picked up first-level dependencies between classes. This indicated that the coupling and the amount of code that was used by each class was higher than could be estimated from the existing documentation. There was also no consistent programming standard, leading to idiosyncrasies between different programmers. Because of the inadequacies that we found in the architecture documentation, and the underestimation of the amount of code used by the potential services, there remained a number of gaps in our understanding of the system.

Code Analysis

To address remaining issues, we first analyzed the code through a code analyzer *"Understand for C++"*.

The code analysis enabled us to validate the input from the contractor and to produce input for the architecture reconstruction tool that would identify dependencies.

From the code analysis, we found that the code was better organized and documented at the code level than most code that we have seen. However, there were inconsistencies in the quality and documentation between different parts of the code that made the analysis complicated.

Architecture Reconstruction

To address the issue of dependencies in more detail, we conducted an architecture reconstruction with a tool called *ARMIN*. Architecture reconstruction is the process by which the architecture of an implemented system is obtained from the existing system [3].

In our analysis, we were interested in

- Dependencies between services and user interface classes
- Dependencies between services and the commercial mapping software
- Dependencies between services
- Dependencies between the services and the rest of the code that mainly represented the data model

The architecture reconstruction was able to identify a substantial number of undocumented dependencies between classes. These will enable a more realistic understanding of the scope of the migration effort if it succeeds.

The architecture reconstruction also enabled us to document the central role of the data model, and to identify it as a potentially valuable reusable component, even though it had not been identified during the initial analysis.

Develop Strategy for Service Migration

In looking at the potential for reuse of the existing legacy components, we found that the current legacy code represents a set of components with significant reuse potential. However, because the current legacy system does not have sufficient architecture or other high level documentation, it was difficult to understand the "big picture" as well as dependencies between different classes. The largest risk in reusing the legacy components concerns the fact that the SOA has not been fully developed. We also recommended that the government organization require the following changes from its contractors to make reuse of its legacy components more viable:

- Suitable set of architectural views
- Consistent use of programming standards
- Documentation of code to enable comments to be extracted using an automated tool
- Documentation of dependencies, especially when they violate architecture paradigms

4. Conclusions and Next Steps

We found that the initial task of determining how to expose functionality as services, while seemingly straightforward, can have substantial complexity. Our conclusions to the client, while not definitive, did point out a number of issues that they had not previously considered. The type of disciplined analysis that we performed appears to have applicability for other organizations that are considering migrations to SOAs.

References

[1] Bergey, J.; O'Brien, L.; and Smith, D. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2). San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.

[2] Brown, A; Johnston, S.; and Kelly, K. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Rational Software Corporation. 2002.

[3] Kazman, R; O'Brien, L.; and Verhoef, C. Architecture Reconstruction Guidelines, 2nd Edition (CMU/SEI-2002-TR-034). Software Engineering Institute. November 2003.

[4] Lewis, Grace and Wrage, Lutz. Approaches to Constructive Interoperability (CMU/SEI-2004-TR-020). Software Engineering Institute. January 2005.

Evaluate Java Program by an Extensible Metrics Reporter

Nuo Li, Jin-liang Ou, Mao-zhong Jin, Chao Liu
Software Engineering Institute, School of Computer Science and Engineering,
Beijing University of Aeronautics and Astronautics, China
Seraphicln, ouj_alonesmoke @hotmail.com   jmz, liuchao@buaa.edu.cn

Abstract

An extensible metrics reporter with visualization environment has been built for Java programs. It is integrated in our "Quality Easy-Software Analysis and Testing" tool (QESAT). Taking into consideration the evolving nature of software development, QESAT is designed with an extensible and maintainable framework in mind. Also extensible is the metrics reporter which is integrated as a plug-in. The metrics gathered can be presented in comprehensive visual graphs, lists or transferred to other understandable formats. In this paper, two case studies are provided to demonstrate how to improve the quality of Java programs using QESAT.

1. Introduction

Software measurement has advanced since the software crisis during the 1960s'. Since then, the need for good software has become apparent. With the popularity of object oriented techniques, software measurement techniques have evolved, such as the CK metrics suite [1][2], MOOD metrics [3], etc. These metrics, though very useful, require more effort to understand. Software engineers would rather review graphics that display important concepts and relationships than comb through lists of metric values [4]. QESAT provides a comprehensive visual environment to evaluate Java software quality with a flexible and extensible architecture. This paper is organized as such: section 2 introduces the metrics suites adopted in QESAT, section 3 details and examines the tool's architecture, and section 4 uses two case studies as examples on how to improve the structure of the program with the metrics offered by QESAT.

2. The metrics suite adopted

Because the basic unit of object oriented programming is a class, we focus mainly on the measurement of the hierarchy and collaboration between classes, as well as the complexity of each class. The Lines of Code (LOC) [5] is used to estimate the program size. In QESAT, LOC is defined as the total number of executable statements defined by Java Grammar. This includes all statements of every method in a class with the exception of declarations and comments. Depth of the Inheritance Tree (DIT) and Number of Children (NOC), which belong to CK metrics suite, examine the inheritance hierarchy complexity. Because the tool focuses on measuring the complexity of programs written by developers, the QESAT root nodes of inheritance trees are the classes offered by J2SE. Response For a Class (RFC) of CK metrics suite measures the communication complexity between classes. Weighted Methods per Class (WMC) of CK metrics suite is employed to measure the logical complexity of a class. And in QESAT, WMC is defined as the sum of McCabe values of all methods defined in a class. For methods, McCabe [6] and Halstead [7] are adopted and adjusted with some definitions specific to Java.

3. Tool design

3.1 Framework of QESAT

The framework of QESAT is a partially complete software system which is intended to be instantiated by concrete functions. It consists of both frozen spots and hot spots [8]. Frozen spots define the overall architecture of the software system which remains unchanged in any instantiation of the framework. Hot spots represent those parts of the framework that are specific to individual software systems. Different from typical frameworks, the QESAT framework supports hierarchical reuse and dynamic extensions.

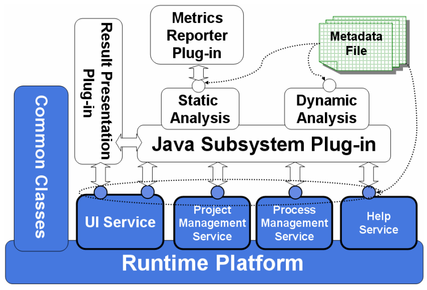


Figure 1. Hierarchical architecture of QESAT

The dark shaded boxes in figure 1 represent the framework of QESAT which is comprised of six parts: Runtime platform, Common classes, and Services (Four parts containing UI, Project management, Process management, and help services). The runtime platform is the kernel of QESAT. Its responsibilities are to boot the system and manage plug-ins and other global system properties. Common classes include utility classes and services are used to group relative hot spots in the framework which are instantiated by plug-ins. At the same time, a plug-in can also be extended by way of other plug-ins by declaring hot spots itself. A plug-in interconnects with the framework or other plug-ins through metadata files. This cuts down the coupling between plug-ins and framework (interpreted in detail by J.L. Ou [9]). In order to avoid dependencies in realization, all plug-ins are physically controlled by the framework while giving programmers a hierarchical view.

The Java sub-system plug-in provides static and dynamic analysis for Java program. This sub-system has been extended by the metrics reporter plug-in. The result presentation plug-in presents the results of the analysis by way of diagrams, charts or tables. It is based on the UI services and provides a monitor for Java sub-system plug-ins by using a well defined XML schema. In order to show the result of the metrics analysis, the result presentation plug-in will transfer the data to the charts or lists. Figure 2 depicts an example of a metrics chart:

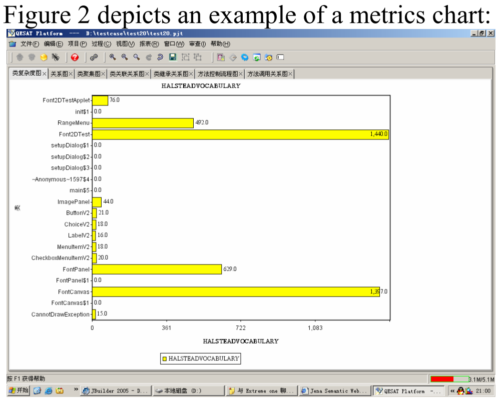


Figure 2. Graphical visualization of metrics values

3.2 Collection of metrics information

Depicted in figure 3, the Java parser processes Java source code to generate a symbol table and Abstract Syntax Tree (AST). The Metrics reporter collects metrics information by moving along the AST and checking the symbol table. Finally, the metrics values are calculated by the metrics reporter. And the information is saved as XML files which will be used for visual presentation.

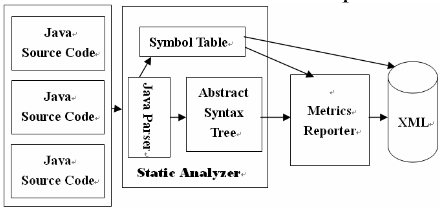


Figure 3. Static analyzer architecture

The Java parser is automatically derived from a parser generated by the Java Compiler Compiler (JavaCC) [10], which is an open source application, provided by Sun MicroSystems. The symbol table is generated by adding actions which record definitions of identifiers to grammar specifications - the input for JavaCC. The Java parser generates the nodes in the AST whose filiations represent the grammar structure of the source code. Based on the information offered by the AST and symbol table, the metrics reporter can extract all types of metrics information. The grammar node is designed following the visitor design pattern [11]. All of the nodes contain a special method to accept visitors.

4. Case studies

4.1 Case study 1

The package measured was named "jstaticdata", which was developed by an inexperienced programmer. The metrics results are presented in a graphical visualization depicted in figure 2. In order to focus on the metrics data instead of the user interface (mainly in the Chinese language), we exported the measurement result files to an Excel document and used them for illustrative purposes in the following discussion.

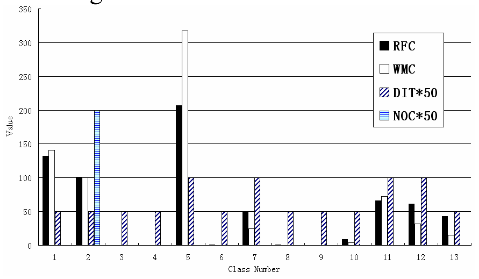


Figure 4. DIT, NOC, RFC and WMC of each class

Figure 4 describes the DIT of all the classes measured. The results show that they are either of values 1 or 2 and that most NOC's are 0. We can assume that the inheritance hierarchy of these classes is not very complex. Notice that the RFC of class No. 5 reaches 207 and its

WMC is 317. Both of them are excessively high in comparison to the other classes. It was discovered that there were 95 methods in this class but some of them implement certain reusable functions that may be invoked by members of other classes. Thus these methods need to be separated from class No. 5 to form a new class. After refactoring, the total number of methods in class No. 5 dropped to 67, the RFC became 124 and the WMC was reduced to 152.

Attention was then focused on the methods themselves in class No. 5. It was discovered that more than half of the methods had McCabe values higher than 4. As most of these methods implemented simple logic, the McCabe values should be less than 4. After analyzing the code, some blocks which implement similar functions were found in many of the methods. Some of the methods were wrapped by other loop or decision statements and worse still, was that if the logic were to change in one block, modifications to all similar blocks would need to be done. Failure to modify all similar blocks would result in hidden logical errors. After refactoring by extracting the blocks to form a new method, the program had become more understandable and maintainable. The metrics results reflected the change and figure 5 shows the McCabe metrics before and after refactoring.

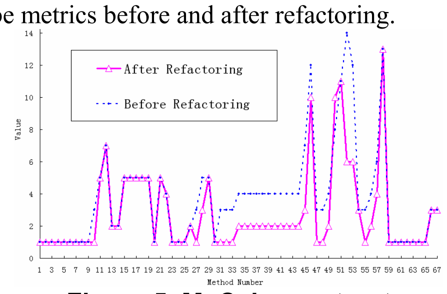


Figure 5. McCabe contrast

Examining McCabe and Halstead values of the methods in class No. 5, it was noticed that methods No. 11, 45, 53 and 58 had relative higher Halstead and McCabe values than the other methods in the class. This alerted programmers and testers to the probability of hidden errors in these methods. Programmers were able to take a pro-active approach and examine the logic used in these methods. However, there are situations where it is impossible to implement simple code. In these situations, the testing of these areas should be stressed.

After refactoring, the RFC of class No. 5 dropped to 121, WMC fell to 187, approximately 40% of the methods fell two McCabe value points lower, and approximately 75% of the methods had shown some decrease in the Halstead value. The large decrease in metrics can be attributed to the fact that the programmer who had written the methods was inexperienced with software development. To support this, it is recorded in our Software Problem Report that nearly 90% of the errors in “jstaticdata” were derived from class No. 5. This proved the intrinsic relationship between the value of class metrics and the number of defects within the class.

4.2 Case study 2

Unlike case study 1, programs measured in this case were developed by 3 programmers (herein referred to as programmers A, B and C). The programmers implemented operations to certain resources respectively, such as add, modify and delete graphs in a database. Programmer A was responsible for various resources including graphs; programmer B dealt with resources involving videos; and programmer C operated the message resources. Although the types of resources were different, basic operations were more or less the same and their metrics were forecasted to be similar. Figures 6, 7 and 8 display the metrics values of classes implementing graphing, video and message operations. The number following every class name is the total number of methods in that class.

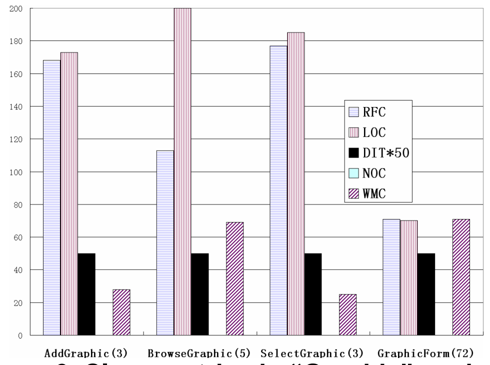


Figure 6. Class metrics in “Graphic” package

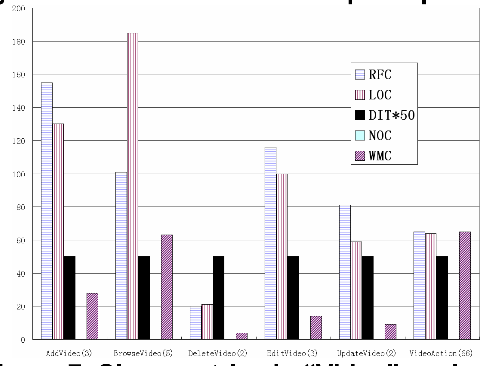


Figure 7. Class metrics in “Video” package

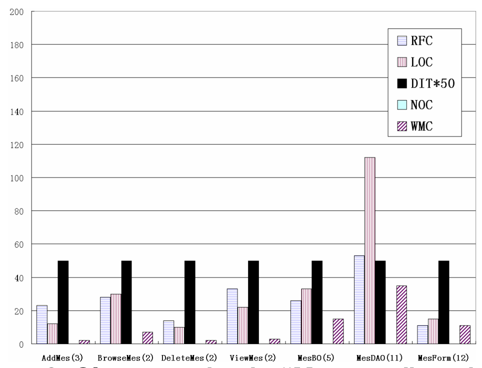


Figure 8. Class metrics in "Message" package

The code in each of the operations is similar in size, though different in structure. Programmer A used the "SelectGraphic" class to enclose all operations of the search criteria while programmer B separated these operations into different classes. Programmer C not only separated the operations, but also split the operations of the database. We can presume programmer C used the DAO pattern [12]. "MesDAO" could provide access to a particular data resource without coupling the resource's API to the business logic. This would have allowed data access mechanisms to change independently of the code which used the data. Although the code of programmer C had more classes than the other programmers, the code was more flexible, extensible and maintainable.

4.3 Summary

Our experiment seeks to attract more attention on the usage of metrics. A workflow may be summarized for improving the quality of a program. First, compute the metrics of relative classes and their methods. Compare the metrics values and any improper structures should be revealed. Developers can then be made aware of which areas are likely candidates for refactoring.

The DIT and NOC values of classes should be checked first. A class with a higher DIT value can reuse more methods; however, there is an increase in likelihood that it is to be affected by other legacy classes. Therefore, such behavior is hard to foresee and classes in this structure are difficult to maintain. The NOC indicates the amount of reuse that is available. A class with a high NOC value may indicate that it is affecting other classes, and thus, must be maintained carefully. In addition to caution, testing in this area should be stressed. Secondly, check the RFC. Classes with a high RFC value will prove difficult to debug as such classes contain, or call, many methods and communicate frequently with other classes. Lastly, compare the WMC values. Classes with an unusually high WMC value relative to the other classes should be focused on and be considered for refactoring. Typically, these modules tend to be more complex, fallible and difficult to maintain. Despite the results in the case studies, real world situations need to be handled realistically. Situations where complex code cannot be avoided should not be blindly subjected to the standard of low metrics values. For programmers to pursue low values at the cost of proper coding would be detrimental to the overall program. To work in situations where high metrics are accepted, an increased focus on testing is required.

References

[1] Chidamber, S.R., and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design", *Conference proceedings on Object-oriented programming systems, languages, and applications*, Phoenix, Arizona, USA, 1991, pp. 197-211.

[2] Chidamber, S.R., and Kemerer, C.F., "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, IEEE Press, USA, Volume: 20, Issue: 6, 1994, pp. 476-493.

[3] Harrison, R., Counsell, S.J. and Nithi, R.V., "An evaluation of the MOOD set of object-oriented software metrics", *IEEE Transactions on Software Engineering*, IEEE Press, USA, Volume: 24, Issue: 6, 1998, pp.491- 496.

[4] Zage, D. and Zage, W, "Module metric signature (MMS) visualization", *Proceedings of 20th IEEE International Conference on Software Maintenance*, IEEE Computer Society, USA, 2004, pp. 512.

[5] Lipow M., "Number of Faults per Line of Code", *IEEE Transactions on Software Engineering*, IEEE Press, USA, Volume 8, Issue 4, 1982, pp. 437-439.

[6] McCabe, T., "A complexity measure", *IEEE Transactions on Software Engineering*, IEEE Press, USA, Volume: 2, Issue: 4, 1976, pp. 308-320.

[7] Halstead M. H., *Elements of Software Science, Operating, and Programming Systems Series Volume 7*, Elsevier, USA, 1977.

[8] W. Pree, "Meta Patterns -- A Means for Capturing the Essentials of Reusable Object-Oriented Design", *Proceedings of the 8th European Conference on Object-Oriented Programming*, Springer-Verlag, Berlin, 1994, pp: 150-162.

[9] OU Jin-Liang , JIN Mao-Zhong. "A Method of Metadata Modeling to Construct Extensible and Flexible Systems"， *Computer Science*, Computer Science Press, China, July 2005 (In press).

[10] Java Compiler Compiler[tm]. https://javacc.dev.java.net/, CollabNet Inc., 2004.

[11] Erich G., Richard H., Ralph E. J., John V., *Design Patterns: elements of reusable object-oriented software*, China Machine Press, China, 2000.

[12] Data Access Object[tm], Sun Microsystems, Inc., 2002, http://java.sun.com/blueprints/patterns/DAO.html.

An Empirical Approach to Software Archaeology[*]

Gregorio Robles, Jesus M. Gonzalez-Barahona, Israel Herraiz
GSyC, Universidad Rey Juan Carlos (Madrid, Spain)
{grex,jgb,herraiz}@gsyc.escet.urjc.es

Abstract

The term "software archaeology" provides a useful metaphor of the tasks that a software developer has to face when performing maintenance on large software projects. The source code of a program at any point in time is the result of many different changes performed in the past, usually by several people, which can be tracked when a version control system is used. We have designed a methodology for analyzing with detail the age of the source code in such cases, and have applied it to several large software projects. As a part of the methodology, we define a set of indexes which can help to characterize the history of a software system, and discuss how those could be used to estimate its past and future maintenance. We also show how our approach to software archaeology is simple both conceptually and computationally, but still very powerful at uncovering useful information.

Keywords: software archaeology, software maintenance, software evolution, empirical analysis

1. Introduction

The idea of applying the concept of archaeology[1] [1] to software maintenance can be tracked at least to the OOPSLA 2001 Workshop on Software Archeology. Software archaeology has been generally used for large old (legacy) systems, but it is valid for any type of software with independence of its age and size. While maintaining a given piece of software, developers have to understand source code that has usually changed many times in the past, producing a result which is the addition of all those changes. If the code is stored in a version control system, its complete history is available, and can be analyzed with appropriate tools. In this short paper, we will focus on the analysis of such a history from a macro point of view, gaining knowledge of the historical structure of a system as a whole, the same way that archaeologists gain knowledge of the history of an ancient city by studying what remains from the different constructions built in it.

For studying projects from this macro-archaeology point of view, we have designed a methodology, which is presented in this paper, and a set of tools to automate it. The methodology starts by determining, using information from the version control system, when and who modified for the last time each line of code. Then, the information for all lines is considered to calculate several indexes which provide useful information about the age of the code, the activity of developers in the past, the level of changes (maintenance), etc. Using this information we may also be able to estimate how much effort new changes would imply.

As case examples of the use of the proposed methodology we have selected nine libre (free, open source) software projects, most of which are among the hundred largest libre software applications included in the latest stable Debian GNU/Linux release[2].

The structure of this paper is as follows. The next section shows the methodology we propose for data extraction and analysis. After that, in section three, we apply our methodology and discuss the results obtained. The forth section introduces a set of indexes that we propose and briefly discuss. Finally, conclusions and further research goals are presented.

[*]This work has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337, by the Universidad Rey Juan Carlos under project PPR-2004-42 and by the Spanish CICyT under project TIN2004-07296.

[1]In American English 'archeology'. The term comes from the Greek meaning 'αρχαιος' (ancient) and 'λόγος' (word/speech).

[2]Debian GNU/Linux is one of the most representative distributions, and probably the largest one. See details in `http://libresoft.urjc.es/debian-counting/sarge`

2. Methodology

To define the methodology, we have considered software projects which store source code in a version control system (in particular, CVS, although it could be easily extended to some other). CVS keeps record of every change in the code. It features a specific option ('annotate') which shows, for any line, the date and author of the last modification.

The process starts by obtaining, for every source file in the current snapshot of the software, the corresponding annotated files. They are stored and parsed. Source files are identified by applying certain heuristics on the file names (for instance, those ending in .c are supposed to be C source files). For considering just code, blank lines and comments are removed also using some other heuristics. In addition, we run some error-correction routines which check for common errors found when mining data from CVS; in order to verify our heuristics, we have compared the number of SLOCs obtained with SLOCCount[3] with the number of lines obtained after applying our heuristics.

Once the annotated files have been parsed, and the mentioned heuristics applied, the resulting data is normalized and inserted into a database, which will be later queried for getting statistical information. This process is performed by a set of scripts which are also responsible for the generation of the kind of graphs shown in this short paper.

3. Case studies

We have applied the described methodology to the code produced by nine libre software projects. They show a great variety from many points of view (age, size, complexity, number of developers, etc.), but all of them are included in major GNU/Linux distributions, which is an evidence of their popularity. In total, our case studies sum up to 9.5 millions lines of code, written mainly in C and C++, and 52,975 source code files. Table 1 presents the most important facts about the code considered.

3.1. Remaining lines

Figure 1 shows how many lines remain untouched since any past date for all the projects relative to the size of each project. The horizontal axis is time, while the

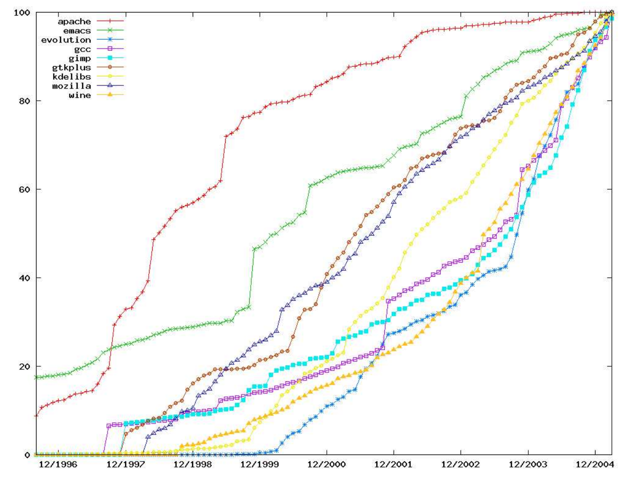


Figure 1. Remaining lines (relative values)

vertical axis is measured in percentages (being 100% the current size of the project). In the figure we can see what which fraction of code is newer than a date. For example, for the case of Apache, approximately 60is posterior to December 1998.

Interestingly enough, the code in all projects is young. Besides Apache 1.3, at least half of the code in all of them is younger than 5 years. Even the code base for Emacs, which we had selected as a legacy system, has a large fraction (up to 70%) which is less than 7 years old.

Apache 1.3 has to be considered separately, since developers are now focused on Apache 2.0, where the main development effort is taking place. However, we expected that at least some corrective maintenance effort would be happening in 1.3, but at least since 2003 that does not seem to be the case.

In the other end of the spectrum, with most of the code being really new, we find GCC, Evolution, GIMP and Wine. in all these cases, this is due, probably, to recent refactorings of the code, including structural and organizational changes.

4. Indexes

To get useful information from software archaeology, it is convenient to use some parameters that help to characterize the history of the project from this point of view. This is the reason why we have defined some indexes that may help to infer some properties of the corresponding development and maintenance process.

[3]We use the '–duplicates' option which counts duplicated files twice as our tools, contrary to SLOCCount, do not filter them out. SLOCCount is available at `http://www.dwheeler.com/sloccount`

Project	Start	Vers. 1.0	Oldest line	SLOCs	SLOCCount	Percent.	Files	Authors
Emacs	(1976)	1985	May 85	974,407	991,552	98.3%	1,522	136
GCC	1985	1987	Sep 97	2,191,764	2,262,632	96.9%	22,349	218
Wine	1993	-	Oct 98	1,033,318	984,710	104.9%	2,201	2
GTK+	1994	Apr 98	(Dec 97)	387,413	389,723	99.4%	839	114
The GIMP	1994	Jun 98	(Dec 97)	548,410	552,473	99.3%	2,244	71
Apache 1.3	1995	Jun 98	Feb 96	82,909	85,758	96.7%	269	51
kdelibs	1997	Jul 98	May 97	605,528	613,742	98.6%	3,131	363
Evolution	1998	Dec 01	May 98	205,278	207,069	99.1%	816	79
Mozilla	(1998)	Jun 02	(Apr 98)	3,414,387	3,510,691	97.3%	19,604	567

Table 1. Summary of the case studies. Columns contain the project name, the year the project started its development, the date of its release 1.0, the number of SLOCs according to our methodology, the number of SLOCs according to SLOCCount, the coincidence for both figures, the number of files, and the authors identified in the current version.

4.1. Definition of the indexes

- **Aging** (measured in SLOC-month). It is a direct measure of how much the software is aging.

$$Aging = \sum_{n=1}^{N-1} lines_n \tag{1}$$

  where n is the month number, being n=1 the first month of the project and N the current one. Notice that the last month is not taken into account.

This index is defined after Parnas' well-known software aging [2] concept, although we only have in mind one of the factors. If we would stick to Parnas' original definition of aging, then we should take into account changes performed on the system, and not only that the software gets old as humans do.

- **Relative aging**. This index makes it possible to compare the *aging* for several projects. It is measured in months and can be obtained from following equation:

$$RelativeAging = \frac{Aging}{lines_N} \tag{2}$$

  where N is the last month considered.

Relative aging represents the amount of time necessary to have the same aging, had the project started with the current number of lines. Of course, it can also be understood as the number of months needed to double the current *a*ging of the project if the system is not touched anymore.

- **Relative 5-year Aging**: relative size to itself as if the project were 5 years old.

$$Rel5yA = \frac{Aging}{60 \cdot lines_N} \tag{3}$$

  where N is the last considered month

Relative 5-year aging allows for easier comparison, defining 5 years as the moment for a system to become 'old'. It is also a needed step for defining the *absolute 5-year aging* index (which will be presented later).

- **Progeria**[4]. As *relative aging* measures the amount of time needed to double the *aging* value, we can compare it to the amount of time needed to double the code base.

$$Progeria = \frac{RelativeAging}{50\% of CurrentCode} \tag{4}$$

Values of progeria lower than 1 are indicative of active maintenance. Projects featuring those indexes have not to fear the consequences of high values of *aging*. However, values above 1 imply that *aging* is growing faster than software maintenance activity and therefore are prone to showing more and more problems.

A new index that provides a value relative to a fixed-size and a fixed-time software system will enable comparison among projects.

- **Absolute 5-year aging**: relative size as if the project had 100 KSLOC and had been started 5

[4]Progeria is a genetic condition which causes physical changes that resemble greatly accelerated aging in sufferers. Source: WikiPedia

Project	Size	Age	Aging	Rel. Aging	Rel5yA	Progeria	Abs5yA
Emacs	974,043	239	62,419,261	64.1	1.07	0.93	10.40
GCC	2,188,033	91	65,558,122	30.0	0.50	0.65	10.93
Wine	1,028,820	78	26,926,319	26.2	0.44	0.80	4.49
GTK+	387,333	88	16,938,898	43.7	0.73	1.04	2.82
The GIMP	540,540	98	16,002,332	29.6	0.49	0.59	2.67
Apache 1.3	82,909	110	6,161,847	74.3	1.24	1.10	1.03
kdelibs	604,888	95	20,089,807	33.2	0.55	1.04	3.35
Evolution	204,951	99	4,796,800	23.4	0.39	0.66	0.79
Mozilla	3,786,735	84	161,394,929	42.6	0.71	1.00	26.90

Table 2. Archaeology indexes for our case studies. Size is given in SLOC, Age in months, Aging in SLOC-month, Relative Aging in months, Progeria, Rel5yA and Abs5yA are indexes.

years (60 months) ago. Serves for comparison purposes among projects.

$$Abs5yA = \frac{Aging}{60 \cdot 100K} \tag{5}$$

where N is the last considered Month.

4.2. Application to the case studies

Table 2 shows how the aging index is not too useful for comparison purposes (although it provides a good idea of the absolute aging). However, relative aging allows for those comparisons. We can see in the corresponding column of the table a summary of the information in figure 1. Apache and Emacs are the systems with the highest relative aging. Evolution, Wine and The GIMP have values in the 20s, which mean that they are still in actively maintained.

With respect to progeria, it can be said that it shows how Mozilla balances aging and evolution, while there are four projects which are becoming old systems: Apache and Emacs (which at this stage of the analysis is not surprising at all), but also GTK+ and kdelibs.

The absolute 5-year aging depends on the size, and has been presented as a proxy of maintainability. It shows that Apache, even having high progeria and aging is still more *friendly* to be maintained than the rest of systems (except for Evolution) because of its small size. Emacs and GCC, even having the latter two times the size of the former, have similar values, while GTK+ and GIMP also show this behaviour.

5. Conclusions and further research

In this paper we have presented an empirical application of the archaeology concept to the macro study of projects maintained in version control systems, with special focus on libre software projects. We have devised a methodology for that study, from which we have defined several indexes which can be used to summarize the development process from the point of view of aging and maintenance.

One of the key findings of this work has been to show that the application of the methodology to the case examples has provided some insight about the maintenance efforts, and the maintainability of the corresponding projects. From a more general point of view, the characterization of a project by several indexes that contribute with useful information about its age and maintainability is probably the key contribution of our work and may help in the decision-taking process by the development teams in libre software projects or by the management team in industrial software companies.

There are many possible future lines of research to explore this approach. First of all, we are looking for better ways of visualization of the archaeological results from a macroscopic point of view. We are also interested in finding relationships with the parameters used in software evolution studies, and in correlating them with effort estimation.

As a summary, we believe that software archaeology provides an interesting framework for digging in the past of a project, so that we can learn patterns and information relevant to infer its future.

References

[1] A. Hunt and D. Thomas. Software Archaeology. *IEEE Software*, 19(2):20–22, 2002.

[2] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, Sorrento, Italy, May 1994.

Constructing a Knowledge Map for a Software Maintenance Organization

Oscar M. Rodríguez-Elias[1], Ana I. Martínez-García[1], Aurora Vizcaíno[2], Jesús Favela[1], Mario Piattini[2]

[1]CICESE, Computer Science Department, Ensenada, B.C., Mexico
{orodrigu | martinea | favela}@cicese.mx
[2]University of Castilla-La Mancha, Escuela Superior de Informática, Ciudad Real, Spain
{Aurora.Vizcaíno | Mario.Piattini}@uclm.es

1. Introduction

Software maintenance requires lots of knowledge. Maintainers must know what changes should do to the software, where to do those changes and how those can affect other modules of the system. Frequently they do not have enough knowledge to make the best decision and must consult other information sources, but these are often unknown or difficult to locate. A knowledge map can help to easily find sources that can be used to obtain the information or knowledge required to perform a specific task; since these maps can be used to point to the sources of specific information or knowledge [1]. This paper presents a work where, qualitative and theoretical research has been applied to develop a knowledge map for a software maintenance organization. This map has been used in the development of a knowledge management prototype that could help software maintainers searching for knowledge and information sources to do their jobs.

2. Identifying the knowledge required by maintainers, and its sources

To build the knowledge map, we investigated and identified the main topics of knowledge that software maintainers require to do their jobs. First we carried out a case study in a software maintenance group to understand the processes and activities performed by the group, the knowledge they require to do their activities, and the sources they use to obtain that knowledge. Then, we performed a bibliographic research to compare our findings to define a more general classification schema for types of knowledge and its sources. The literature review was based on research papers, such as case studies, software maintenance ontologies, standards for software engineering and maintenance, and the SEWBOK [2]. However, we used the case study to focus and establish the basis of a practical classification for the group studied. Next, we describe how the knowledge map was developed.

3. Constructing the knowledge map

To construct the knowledge map we first defined a classification schema and structure the types of knowledge and knowledge sources. The classification schema was used to define a metamodel which describes the relationships of the knowledge subjects and its sources. Then we defined a template to describe specific knowledge subjects and sources. These descriptions were used to construct the knowledge map by representing knowledge subjects and sources into a XML format. Next, we present this development.

3.1 Classification of the types of knowledge and their sources

Knowledge subject classification

The classification of knowledge subjects was done following a schema that consists of three levels of abstraction: ***categories, knowledge areas*** and ***knowledge subjects.*** At the first level are ***categories***, which are structural elements of high level abstraction used to classify related areas of knowledge. A category can also contain more specialized categories. In the second level are the ***knowledge areas***, which are subdivisions of the *categories* that are logically related with them, for example by aggregation or composition. An *area* can contain more specialized sub-areas or ***knowledge subjects***. The *subjects* represent basic concepts with an explicit and well defined description. They are used to describe knowledge about a set of elements that can be considered as a unit. However, a *subject* can also be composed by more detailed *subjects*. Following the schema just described, we defined some general areas of knowledge grouped into three main categories: (I) the ***knowledge related to the software maintenance process***; (II) the ***knowledge required for the organization's life***; (III) and the ***general knowledge that is not part of the other two categories***.

The ***knowledge category of the maintenance activities*** is the most important, and most of the areas of knowledge are grouped here. This category is composed by three subcategories: 1) *Computing fundamentals* contains areas of general knowledge about computing, such as operative systems, programming languages, etc.; 2) *Software engineering* considers the knowledge related with the phases of the software development life cycle, such as project management, analysis and design, etc.; and 3) *Application knowledge* category, groups the knowledge related with the applications maintained by the team, such as specific knowledge about the products, for instance the architecture, structure, functionality, history, etc.; and the knowledge of the domain that the applications support.

The organization's life knowledge category considers the knowledge that is not directly related to the activities of software maintenance, but that all employees must know, such as the structure, norms and policies, goals, etc. of the organization; and knowledge about other processes followed by the organization.

Finally, in the ***general knowledge category***, the knowledge and skills that are not part of the daily work, but can be useful for special purposes, are considered. For example, foreign languages speaking and writing, group work coordination, leadership, etc.

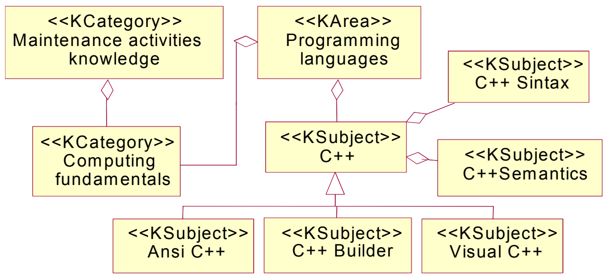

<<KCategory>> Maintenance activities knowledge
<<KArea>> Programming languages
<<KSubject>> C++ Sintax
<<KCategory>> Computing fundamentals
<<KSubject>> C++
<<KSubject>> C++Semantics
<<KSubject>> Ansi C++
<<KSubject>> C++ Builder
<<KSubject>> Visual C++

Figure 1. Knowledge subjects classification example.

Figure 1 illustrates an example of the knowledge subject's classification schema. This example shows how a specific programming language, such as C++, has been classified into a programming languages area, which is grouped into the computing fundamentals subcategory that corresponds to a part of the knowledge required for the activities performed by the members of the organization, in our case, the software maintenance activities.

Knowledge sources classification

A schema composed of categories and types of sources was used for classifying the sources of information and knowledge.

Sources of knowledge were divided into four categories: *(I)* ***documentation,*** *(II)* ***people,*** *(III)* ***the maintained systems' elements, and*** *(IV)* ***support tools***. Next we describe each of these categories.

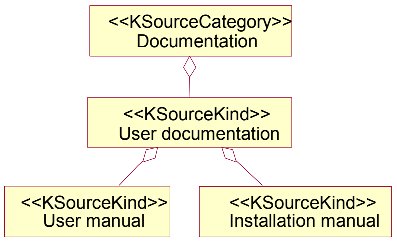
Documentation category groups all the kinds of documents that can be used by the maintenance group. These documents were classified into six main types: 1) *System documentation,* 2) *Technical documentation,* 3) *User documentation*, 4) *Organizational documentation*, 5) *Maintenance process documentation*, and 6) *Other documents.*

The people category refers to all the persons that are consulted by the members of the maintenance group. This category has been divided into three: 1) *Users/Clients* (Even though users and clients play different roles, we have decided to take them as a single category since in the group studied there is not a clear separation between the roles played by users and clients [5]). 2) *Staff members* are all the persons working in the maintenance group; and 3) *Other experts* refers to all the persons that are not staff members or users, but that are consulted by maintainers to obtain specialized knowledge, such as knowledge about the application domain, a specific programming language, etc. These experts can be either internal or external to the organization. For example, some maintainers consulted friends that are not in the organization, or consulted experts through internet newsgroup, email lists, etc.

The system category refers to all the elements that constitute the products that are being maintained and that can be sources of information and knowledge. These elements have been divided into three types: 1) *Executable system*, 2) *Source code*, and 3) *Data bases of the systems maintained*.

Finally, the ***support tools category*** is concerned with all the tools used by maintainers to obtain information or knowledge. These tools have been divided in two types: 1) *Maintenance activities support tools* are those used for supporting activities of the maintenance process; and 2) *General support tools* are those that are not directly related to the maintenance activities. For example, organizational memories or portals, content management systems, document repositories, etc.

Figure 2 presents and example of the knowledge sources classification schema; where two types of documents with information directed to the users of the applications maintained (the user and installation manuals) are classified as user documentation; which is a subdivision of the documentation category.



<<KSourceCategory>> Documentation
<<KSourceKind>> User documentation
<<KSourceKind>> User manual
<<KSourceKind>> Installation manual

Figure 2. Knowledge sources classification example.

Knowledge topics and sources metamodel

A metamodel was defined based on the classification schemas to help define the relationships between the topics of knowledge, their sources, and the activities where knowledge and sources are required, generated or modified. Figure 3 shows the general view of this metamodel. The topics and sources of knowledge are considered as knowledge concepts. These knowledge concepts are used and can be generated or modified in the processes, activities or tasks performed by maintainers; defined as work definitions (concept took from the SPEM specification, and that refers to a kind of operation that describes the work performed in a process [3]). Each work definition has a purpose, described as a goal. Each source of knowledge can also have information or knowledge about topics or other sources. The levels of experiences or details about the knowledge and information can be defined with *KLevel* elements. Finally, sources of information can have a location where they are consulted, such as physical address, email, electronic address, etc.; and a format, for example pdf, word, or excel for electronic files.

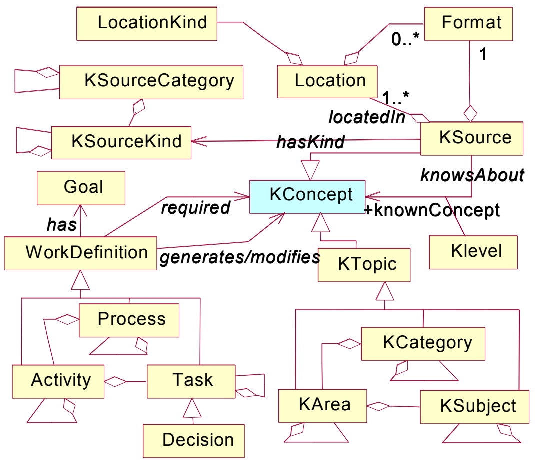


Figure 3. Metamodel of knowledge and sources.

Following the metamodel, we defined two templates to describe specific topics and sources of knowledge. The templates and their use are described next.

3.2 Describing knowledge subjects and their sources

To describe knowledge subjects we followed the template exemplified in Table 1. In this template, each subject is identified by a name and a short description. Then, the main cognitive and technical knowledge related to the subject are defined. Cognitive knowledge refers to *know what*, for example, which activities we must do, what information is required to do these, where we can find that information, etc. Declarative or cognitive knowledge can be divided in two types [4]: 1) topic knowledge, that refers to knowledge about concepts, their definitions, properties, and relationships; and 2) episodic knowledge, that represents the experiences on the use of knowledge. Finally, procedural or technical knowledge helps to know how an activity should be done.

The example of Table 1 refers to the knowledge related to an activity. The *topic knowledge* items can be used to identify sources that can help to obtain information about the topics defined. The *episodic knowledge* items establish the situations that can cause generation of knowledge related to the subject. These definitions can be used to identify people that have been involved in one or more of these situations. *Procedural knowledge* definitions can be used to identify sources of knowledge that can be useful to obtain information about how to do something related to the subject.

Table 1. Example of a knowledge subject description.

Knowledge subject	Performing modifications to the check bills elaboration module in the finances system
Description	Knowledge about the check bills elaboration process in the finances system, and about how to modify the module
Topic Knowledge	Which is the module of check bills elaboration
	Which are the surce files of the module
	Where are those files
	Related modules
	Which is the programming language used
Episodic Knowledge	Experience using the check bills elaboration module
	Experience developing the check bills elaboration module
	Experience modifying the check bills elaboration module
Procedural Knowledge	How to access the module
	How to make changes in the module
	How to identify problems in the module
	How to correct problems in the module

The sources of information and knowledge are described by templates as the showed in Table 2. Each source has a unique identification (id), a short description, and it is classified in a category and a kind. Each source can be consulted, at least, in one location. Locations are defined by its type and a description that is used to provide specific access data. Depending on the location, the source can have a physical support (such as paper, video tape, CD, etc.) and a format; for example, the source described in Table 2 is an electronic file, a word 2000 document. Finally, the main information and knowledge subjects that can be obtained from the source can be specified into the "*knows about*" list. This information is later used to identify in which activities the source can be helpful.

Table 2. Example of a knowledge source description.

Source id:	p1230_requerimientos.doc		
Category	Documentation		
Kind	System documentation / Requirements		
Description	Document containing the requirements specification of the finances system SIREFI		
Located at			
Kind	**Description**	**Physical support**	**Format**
Electronic file	Directory: "c:\projects\p1230\documentation\" on project files server		word 2000
Knows about			
Concept		**Level**	
Requirements of SIREFI system		Advance	

The templates were used in the initial phase of our work. The resulted knowledge map has been used in a prototype of a KM system. An example of how the system uses it is presented in the next section.

4. Using the knowledge map

The prototype of the KM system is based on a multi-agent architecture where there is a *staff agent* that plays the role of assistant of a member of the maintenance team [5]. When the staff agent detects that the maintainer is performing an activity, it tries to identify the knowledge required by that activity. For example, if the maintainer wants to solve a problem reported, the agent obtains information from the problem report, such as the system and the module where the problem appeared, the type of problem, etc.; then, it tries to infer what knowledge can be required to solve that problem; for example, which are the source code files of the module, where they are, etc.

Once the agent finishes defining the list of subjects of knowledge, it starts searching for knowledge sources that could have information about the subjects defined. When the search is finished, the agent informs the user that there are sources of information that can be relevant to the activity being done. If the user decides to consult those sources, the system shows a window with the sources found, grouped by types. When the user chooses one of the sources, the system shows information such as how or where that source can be consulted, and the main subjects of knowledge related to the activity, that can be obtained from it.

The prototype and the preliminary knowledge map where tested following scenarios obtained from the case study carried out [5]. The knowledge map was developed with information obtained from the group studied, and represents real situations.

5. Conclusions and future work

Finding methods and tools that help software maintainers reduce the time needed to do their jobs can provide major benefits to software organizations; for example, by helping maintainers reducing the time they spend searching for sources of information to obtain the knowledge they need to perform their jobs. In this paper we presented how we developed a knowledge map for a software maintenance team, by classifying the main knowledge required by the members of the team and the sources of information available. This map also helps to identify where that knowledge and sources can be required by defining the relationships between the types of knowledge, the sources, and the main activities performed by the team.

We have used the knowledge map in a prototype of a KM system. However, more research must be done to measure how useful the map could be in a real environment. In order to make that research, a more complete knowledge map should be developed and adapted to the maintainers' work environment.

Acknowledgements

This work is partially supported by CONACYT under grant C01-40799 and the scholarship 164739 provided to the first author, and the MAS project (grant number TIC2003-02737-C02-02), Ministerio de Ciencia y Tecnología, SPAIN.

References

[1] T. H. Davenport and L. Prusak, Working Knowledge: How Organizations Manage What they Know. Harvard Business School Press, Boston, Massachusetts, 2000.

[2] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. L. Tripp, "SWEBOK: Guide to the Software Engineering Body of Knowledge," IEEE Computer Society, Los Alamitos, CA., 2004.

[3] OMG, "Software Process Engineering Metamodel Specification (SPEM)," Object Management Group, 2002.

[4] P. N. Robillard, "The Role of Knowledge in Software Development," CACM, vol. 42, 1999, pp. 87-92.

[5] O. M. Rodríguez, A. I. Martínez, J. Favela, A. Vizcaíno, and M. Piattini, "Understanding and Supporting Knowledge Flows in a Community of Software Developers", LNCS 3198, 2004, pp. 52-66.

Incremental Product Line Modelling

Serguei Roubtsov and Ella Roubtsova
Eindhoven University of Technology,Den Dolech 2, P.O.Box 513 5600 MB The Netherlands
S.Roubtsov@tue.nl E.Roubtsova@tue.nl

Abstract

A traditional software product line approach struggles with complexity and weak evolution support. We propose an incremental product line approach based on controllable inheritance of product model specifications. We use hierarchies of inherited product specifications accompanied by correctness control of product model transformations. An industrial case study from the embedded systems domain is provided to demonstrate the approach.

1. Introduction

Software product lines (SPL) employ an architecture-based methodology of software system development. It starts by choosing a set of products comprising a product line and proceeds by identifying what requirements are common to all products (commonalities) and what product features make them different (variabilities). Commonalities between SPL members are captured by a generic architecture. Variabilities are usually introduced into this architecture by means of variation points, which imply unresolved diversity in the generic and component architectures that should be explicitly introduced and bound into a concrete product during product line member development.

A common SPL architecture with variability management fulfils a double role. Firstly, it provides the *reference of integrity* for SPL component reuse. Secondly, the diversity of all product line members, existent or future, should correspond to the variability already implicit in the generic architecture. So, the SPL architecture should provide *correctness* of product modifications.

However, there are some disadvantages of such an architecture-centric approach. The first problem is complexity. Among other tasks design of the reusable SPL architecture is an especially complicated problem. The more variability is introduced into the architecture, the more benefits of reuse should be expected. However, design of such a flexible architecture meets a truly challenge.

The second problem is evolution support. Requirements are changed, technology is improved. It is very hard to predict the features and, therefore, the architectures of future product line members.

The possible alternative is component-based software development. It implements component modification and composition instead of architecture-based variability management. Similar implemented products are reused with extensions which are required for a new product. However, in the absence of a fixed common architecture the problems of SPL integrity and product design correctness rise sharply. Component modification and composition rules are static, they do not guarantee that the entire system behaviour comprises the behaviour of composition parts in a correct manner. The evolutionary approach needs a design methodology that can help designers collect useful features of already implemented SPL members and avoid incorrect design decisions while they introduce new product functionality.

We propose an evolutionary software product line modelling method based on the inheritance of product design specifications and correctness control of model transformations. In our approach design specifications are implemented using a UML profile with defined inheritance relations on specifications. The profile includes a special type of UML class diagrams, interface-role diagrams. Component system behaviour is specified in the profile using UML sequence diagrams. Process semantics is used as a basis for inheritance relations on component behavioural specifications [2].

Correctness control is provided by product model transformation checks using bisimulation inheritance of processes [1]. Applying of backward derivation rules to produce a parent process specification from an inheritor's one allows a designer to prove correctness of inheritance or to find the points of wrong design decisions.

The next section describes a case study from the embedded systems domain. Section 3 explains our method and provides illustrations using the case study. That section also contains a conclusion and some observations about future work.

2. Case Study: Scientific Silicon Array X-Ray Spectrometer

Our case study is a product line representation of Scientific Silicon Array X-Ray Spectrometer (SIXA) Control Software [1]. This is an onboard satellite system that provides scientific data in two measurement modes: Energy Spectra (EGY) and Single Event Characterization (SEC). There are several variants of SIXA spectrometer with different features. We intend to model two members of the SIXA software product line: *stand alone EGY Controller* and *combined EGY and SEC Controller*.

The SIXA Controller fulfils the following *functional requirements*. It 1) receives measurement programmes from the ground via a satellite computer, 2) provides data measurement, 3) collects and sends data back.

The system comprises four interconnected subsystems:

- *Measurement Control* subsystem. This subsystem provides *Controller Commands* interface with an onboard satellite computer. External control commands and measurement programmes come via this interface.
- *Data Acquisition* subsystem. It executes measurement programmes received via its interface *Control Data Acquisition* from *Measurement Control* subsystem.
- *Data Management* subsystem. It fills its internal buffer with data received from *Data Acquisition* subsystem via interface *Save Data* and sends scientific data back to the ground via *Satellite Computer* interface *Controller Data Response* following commands from *Measurement Control* subsystem via interface *Control File Management*.
- *Satellite Computer* that is regarded as an external system. It uses Spectrometer interface *Controller Commands* and receives scientific data via its own interface *Controller Data Response*.

The described above functionality is common for the entire SPL. The variability is defined by the different measurement modes that have to be implemented. EGY and SEC modes are realized by different *Data Acquisition* subsystems and corresponding interfaces *Control Data Acquisition* and *Save Data*. There is another difference: EGY Controller *Data Management* subsystem sends data to the satellite computer after a measurement programme has been fulfilled completely, whereas SEC Controller *Data Management* subsystem is able to initialize data exchange when its internal buffer is full. So, that subsystem should be able to send such a request to *Satellite Computer*.

1 We thank Prof. Eila Niemela and Tuomas Ihme from VTT Electronics, Finland for sharing the insights into this case study

EGY and SEC Controller has to provide functionality of each stand alone mode whatever has been chosen by the ground measurement programme.

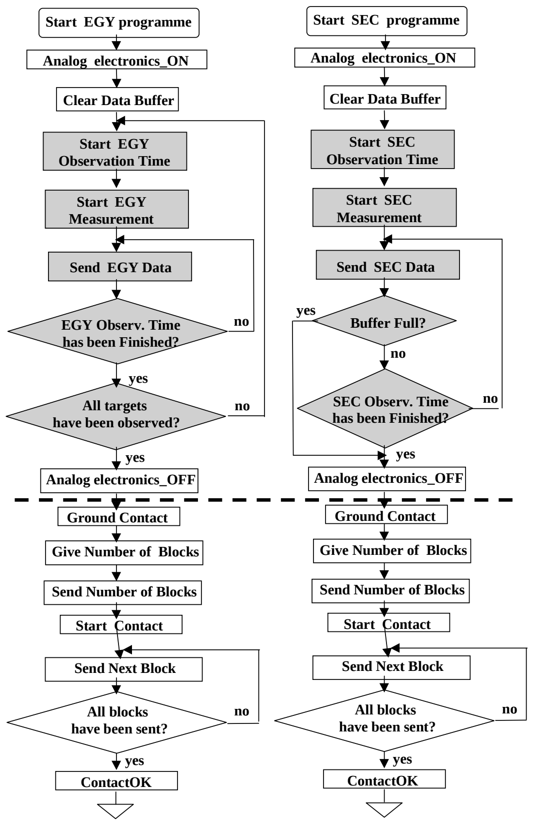


Figure 1. Observation algorithms for SIXA Spectrometer. On the left hand side: EGY mode; on the right hand side: SEC mode; measurement sub-process is above – – –– line; data exchange sub-process is below.

The *behavioural requirements* to the SIXA Spectrometer software are defined by two data observation processes, one process for each observation mode. Both processes comprise two sequential sub-processes: data measurement and data exchange. Using usual algorithmic notation the processes can be described as it is shown in Fig. 1. Each block in Fig. 1 corresponds to an operation call that is performed by interacting SIXA Controller software subsystems and supported by hardware signals.

The data exchange sub-process is common for EGY and SEC modes. The data measurement sub-processes are partially different. The dark blocks in Fig. 1 depict the steps of the measurement sub-processes which are different for EGY and SEC modes.

3. Incremental Product Line Modelling Method

The method comprises two parts: product model specification and the definition of inheritance of product specifications with the derivation rules allowing to prove correctness of model transformations.

The *product line member specification* is a pair $PrSp = (IR, BS)$ where IR is an interface-role specification and BS is a behavioural specification.

The interface-role specification describes static aspects of product functionality. Roles can *provide* interfaces, which other roles can *require*. Each such a pair of roles interacting via an interface can model a piece of product functionality, i.e. a product feature. So, product functional requirements can be mapped directly to interface-role specifications.

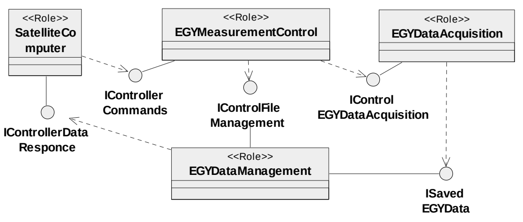


Figure 2. Interface-role diagram for EGY Controller

The interface-role specification is realized in the UML profile [2] and presented by a UML class diagram, where roles are UML classes with stereotype $\ll$Role$\gg$ and interfaces are classes with stereotype $\ll$Interface$\gg$. Interfaces are depicted by cycles. Provided relations are presented by UML realize-relations between roles and provided interfaces and depicted by solid lines. Required relations are the same as UML dependency relations between roles and required interfaces. A required relation is depicted by a dashed arrow directed from a role to a required interface. The interface-role diagram of EGY Controller is shown in Fig. 2.

The behavioural specification describes dynamic aspects of product functionality. A grain of product behaviour is presented by a *pair of actions* [2]. The first action of the pair is an *operation call*, the second one is an *operation return*.

As a result of product IR specification, action set A_{PrSp} is introduced for the entire product specification: $A_{PrSp} = \{a_1, a_2, ...\}$.

Using action set A_{PrSp} we construct behavioural specification BS as a finite *set of sequences* representing product behavioural patterns [2]: $BS = \{S_1, S_2, ..., S_n\}$, where $S_i, \forall i = 1, 2, ..., n$ is a *sequence of actions* $a_j, a_k \in A_{PrSp}, \forall j, k = 1, ..., |A_{PrSp}|$: $S_i = \{a_j, a_k, ...\}$. The definition means that we can construct behavioural pattern S_i using any action from action set A_{PrSp} any number of times.

The behavioural specification is realized in the UML profile [2] and presented by a set of UML sequence diagrams, one diagram for each sequence S_i.

We regard *inheritance of product line members* as inheritance of product behaviour. If, for example, product EGY and SEC Controller inherits product EGY Controller, then it inherits the possibility to observe energy spectra and extends it by the SEC spectra observation facility.

Behaviour specification $BS_q = \{S_{1_q}, S_{2_q}, ..., S_{n_q}\}$ *inherits* $BS_p = \{S_{1_p}, S_{2_p}, ..., S_{m_p}\}$ *if* $n \geq m$ *and each sequence* S_{i_q} *inherits corresponding sequence* S_{i_p}. Each sequence S_i is defined by set of actions A_{PrSp} and this set is defined by the set of required relations on product interface-role specification IR.

Inheritance of roles is defined in the UML profile [2] and corresponds to the specialize-relation between UML classes. The relation is shown on the interface-role diagram by a solid line with the triangle end $\dashv\!\triangleright$ directed from role-child to role-parent. The interface-role diagram of EGY and SEC Controller is shown in Fig. 3.

As a result of inheritance, the child interface-role specification comprises two parts: $IR_q = (IR_q^{Inh}, IR_q^{New})$, where IR_q^{Inh} contains inherited roles, their provided interfaces and provided relations, and, possibly, required relations; IR_q^{New} is a new part, which contains new roles, interacting via new interfaces; it realizes new product functionality and inherits the functionality of a parent product.

The inheritor EGY and SEC Controller has to utilize functionality of EGY Controller and extend it by new SEC Controller functionality. New functionality is realized by three new interfaces of the child roles (Fig. 3).

To define inheritance of product behaviour we apply process semantics on behavioural specifications BS. Following [1] we use a process semantics of type $P = (A, \mathcal{P}, T)$, where: A is a finite set of actions; $\mathcal{P} = \{p, p_1, p_2, ..., p_F\}$ is a finite set of abstract states from initial state p to final state p_F; T is a set of transitions. Transition $t \in T$ defines a pair of states (p', p''), such that p'' is reachable from p' as a result of action $a \in A$: $p' \overset{a}{\Longrightarrow} p''$.

Considering set of actions A as set A_{PrSp} we construct a single *process graph* for the entire product behaviour specification. Each finite sequential path on this graph corresponds to sequence S_i from product behaviour specification BS.

The process graph for EGY Controller is shown in Fig. 4 a). It contains the only sequential path. The behaviour specification for EGY and SEC Controller contains three se-

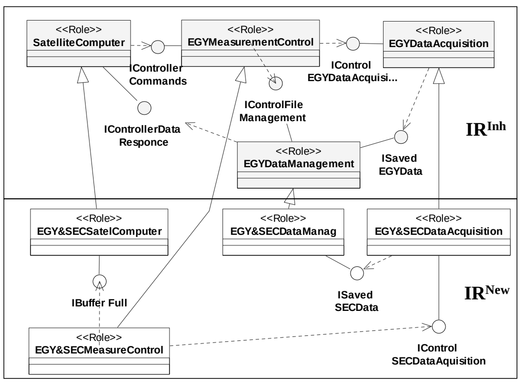
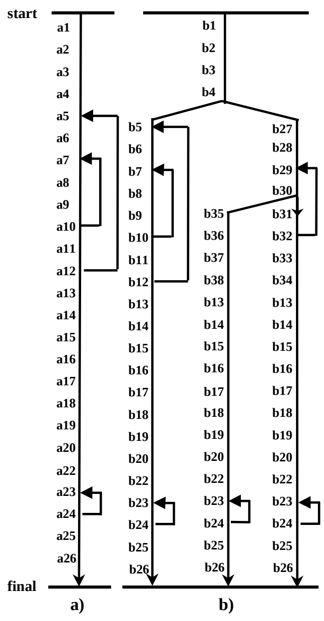


Figure 3. Interface-role diagram of EGY and SEC Controller

quences realizing the requirements to the behaviour of the second product. The corresponding process graph for EGY and SEC Controller is shown in Fig. 4 b).



start
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19 a20 a22 a23 a24 a25 a26
b1 b2 b3 b4
b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16 b17 b18 b19 b20 b22 b23 b24 b25 b26
b35 b36 b37 b38 b13 b14 b15 b16 b17 b18 b19 b20 b22 b23 b24 b25 b26
b27 b28 b29 b30 b31 b32 b33 b34 b13 b14 b15 b16 b17 b18 b19 b20 b22 b23 b24 b25 b26
final
a)
b)

Figure 4. Process graphs for a) EGY Controller; b) EGY and SEC Controller

As a result of inheritance of interface-role specifications action set A_{PrSp_q} of the inheritor contains two subsets: one is a subset of actions, which are realized by *inherited required relations from* IR_q^{Inh} and the other is a subset of actions, which are realized by *newly designed required relations from* IR_q^{New}.

The short format of the paper does not allow us to go in detail about applying the derivation rules [3] which prove correct inheritance of behaviour specifications. Those rules allows us to apply the *life-cycle bisimulation inheritance* [1] on processes of type P to process graph representation and define the conditions of correct inheritance of product specifications. Briefly, if the parent action set contains only inherited actions we, using the blocking action δ, eliminate from the child process graph all alternative branches that are started by new actions. Next, we apply the hiding action τ and eliminate the rest of new child actions. If the resulting transformed graph is equal to the parent graph with the renamed corresponding actions, then the child specification is a correct inheritor of the parent specification.

In our case study EGY and SEC Controller is a correct inheritor of EGY Controller. Indeed, if we rename parent actions $\{a1, a2, ..., a26\}$ to $\{b1, b2, ..., b26\}$ and hide and block the new actions from the child set, the child process graph is transformed to the parent one (actually, for such transformation blocking of action b27 in Fig. 4 b) is enough).

In case of incorrect inheritance of a parent specification a transformed child process graph contains not eliminated τ and δ actions. The rest of a sequence starting by such an action becomes unreachable [1]. The positions of τ or/and δ actions show the points of design errors. These errors correspond to the actions, which cannot be executed within a given specification. Thus, the method allows a designer not only to prove correctness of inherited specifications but also to find design flaws.

In this paper only one type of behaviour inheritance - life-cycle bisimulation inheritance - is described. In paper [3] we argue that there is no single notion of behaviour inheritance, there is an infinite set of them and it is up to the designer to decide which one to choose. The choice has to be imposed by requirements as a constraint on the system in hand. It can be shown [3] that different types of component system inheritance can be proved using similar methods as the one described in this paper. In future work we intend to find the application of the notion of a behaviour inheritance constraint to SPL design.

References

[1] Basten T., W.M.P. van der Aalst. Inheritance of behaviour. *The Journal of Logic and Algebraic Programming*, 46:47–145, 2001.

[2] Roubtsova E. and R. Kuiper. Process Semantics for UML Component Specifications to Assess Inheritance. *ENTCS, 72,3 Elsevier Science Publishers, http://www.elsevier.nl/gej-ng/31/29/23/127/48/show/Products/notes/index.htt*, 2003.

[3] Roubtsova E.E., S.A.Roubtsov. Constrants of Behaviour Inheritance. *Proceedings of the First European Workshop on Software Architecture. Springer LNCS 3047*, pages 115 – 134, 2004.

A Study of Evolving Complexity in a Re-Structured Business Application

M.P. Ware & F. G. Wilkie
Centre for Software Process Technologies,
University of Ulster, Newtownabbey, Co Antrim. BT37 0QB. Northern Ireland
(mp.ware@ulster.ac.uk, fg.wilkie@ulster.ac.uk)

Abstract

This paper is concerned with an examination of the appropriate alternative courses of action when identifying overly complex components within a business application. A study is described which uses complexity metrics drawn from a number of sources including the Chidamber and Kemerer (C&K) metric suite and general application specific descriptive statistics. The metrics were applied to a commercial product implemented in C++. They were used in tracking complex components across two major releases of the application when determining the effectiveness of responses to these components. A discussion is presented which assesses the benefits of alternative courses of action available to a project manager on detection of complex classes. The conclusion to this paper includes observations on our potential ability when seeking to control application complexity.

1. Introduction

Measures relating to an application may be used in attempts to predict and manage change. Such measures tend to include maintenance related statistics for example number of incidents, closure rates, time to close and effort. These measures are apt to be contrasted to metrics relating to the applications architectural and implementation features; measures of inheritance, size, coupling and cohesion. Many studies exist which validate the use of software metrics for predictive purposes [1], [6], [4], [8]. The majority of these studies assess the predictive power of the metrics against maintenance data usually suggesting that high complexity is correlated with fault or change proneness, and a lack of potential for application extensibility or reuse.

The underlying implication of these studies is that concentrations of complexity in an application are bad and to be avoided. Consequently the antithesis is that simplicity is good and to be promoted.

This view is expressed in one of the main tenets of object oriented technology – abstraction. Booch [3] writes that 'abstraction is one of the fundamental ways that we humans cope with complexity'. The principle of abstraction pervades the various divide-and-conquer techniques used in object oriented design and development; such as model based development, assessment of viewpoints, dynamic and static modelling even the primitive constructs of class, function and attribute.

If a project manager is using complexity metrics to monitor an application then surely the manager will respond to the detection of high or growing measures. Many metric analysis tools and metric studies focus upon the detection of complexity, however less work has focused upon the correct response to this. We can ask

1. What should a project manager do when receiving measures which indicate relatively high levels of complexity within an application?
2. Is the objective to re-engineer for a reduction of measures and the promotion of less complex components within the application?
3. Is it possible to achieve simplicity within inherently complex systems?

An assessment of the answers to the above questions should help the manager when deciding upon the appropriate course of action on identifying application complexity.

Section 2 presents a study which uses metrics to indicate complex features pertaining to the first release of an application. We track these features across subsequent re-design and maintenance activity. In Section 3 we assess relative levels of complexity related to and re-design alternatives and in section 4 we present conclusions derived from the issues raised.

2. Empirical Study

The Centre for Software Process Technologies (CSPT) is dedicated to promoting best practice within the local software industry. It has an on ongoing programme of research into the application of software metrics. An automated code analysis tool [7], [9] is used to harvest various measures from two successive versions of a large commercial application written in C++. The application was subject to a major re-design effort dedicated to the promotion of operating efficiency, systems maintainability, extensibility and general robustness. This study also employs associated product maintenance data and feed back from team members.

The metrics used in this study primarily spring from the C&K metric suite [5] and include additional metrics developed by the Centre [10], [11]. Maintenance related measures pertain to class revision counts, code lines added and deleted. A description of measures is listed in Table 1.

A survey was conducted of two releases of the application. The initial version consisted of 107 concrete classes implementing 1795 methods with between 0 and 82 methods per class. The maximum depth of inheritance was 4 and only 9 classes did not participate in an inheritance hierarchy.

Table 1. Brief Definition of Measures

Coupling Between Objects – CBO	Number of distinct classes to which the subject class is coupled via method invocation, class attribute declaration, method parameter declaration or local method data item declaration.
Response for a Class - RFC	Distinct count of number of interface methods for a class added to the number of methods that the interface methods may call.
Coupling Complexity Forwards - CC_F	Measure of the fan-out of class. A distinct count of the number of interface methods invoked by the subject on external classes.
Coupling Complexity Indirect - CC_I	Measure of the internal coupling of a class. A distinct count of methods participating in the internal class call chain.
Weighted Measures per Class Methods - WMCM	Size measure, a count of all the methods in a class.
Weighted Measures per Class – Lines of Code – WMCLOC	Size measure, count of all lines of code representing method implementation for the class.

The second version consisted of 112 concrete classes possessing 1807 methods with between 0 and 81 methods per class. The maximum depth of inheritance was still four although the inheritance hierarchy had been revised, 16 classes did not participate in any hierarchy. The only measure showing a decline was WMCLOC which relates to lines of code. Correlations associated this measure with increased service provision and service utilisation suggesting that the semantics of each class was being refined.

2.1 Strategy

Pareto analysis is based upon the observation that many defects will have a small number of causes '80% of the defects come from 20% of the modules' Boehm [2]. Pareto analysis is therefore used to detect complex classes in version 1 of the application; these classes are ranked according to measure and tracked in version 2 of the application. In addition classes from version 2 of the application with emerging high measures are included in an impact analysis focusing upon the relative ranking of classes Table 3.

2.2. Managing Complexity within an Application

The modifications applied to version 1 of the application impacted upon the complex classes, however an overall decrease in application complexity was not observed. The figures presented in Table 3 indicate that change within the application did affect the ranking of measures; it is however the degree of the impact that varies. According to ranking the majority of classes remained complex (52 – 71%). A high proportion of classes decreased in complexity ranking (37 – 58%) but not sufficiently, so as to remove them from the list of most complex classes in the application. In addition to classes decreasing in ranking a small but significant number of classes were removed from the application (11 – 22%). Their functionality was consolidated and implemented in new classes. A significant proportion of classes increased in ranking (22 – 42%) and a few classes maintained a stable rank (0 – 11%).

Of the classes that were new to the list of complex classes a significant number were classes new to the application. Factoring out, replacement and addition of classes did not necessarily reduce complexity.

3. Considerations when Managing Complexity

Given a high measure there are essentially three choices that a project manager can make when directing maintenance activity.

Table 3. The impact of inter-release change activity upon relative complexity ranking

	% of Classes Removed	% Decrease in Rank	% Stable Rank	% Increase in Rank	% Remain Complex	% New to Rank	% New to App.
CBO	17	50	0	33	71	29	14
RFC	11	42	11	37	67	33	14
CC_F	22	56	0	22	50	45	9
CC_I	18	47	6	29	60	45	15
WMCM	11	37	11	42	67	33	14
WMCLOC	11	58	0	32	52	48	24

A. To stand back from actively managing the situation thereby allowing an organic growth or decline in the measure.
B. To consciously limit measures by techniques such as the sub-division of complex components.
C. To promote fewer components but accept greater complexity within them.

Each alternative suggests a potential growth in complexity. Option – (A) may produce either a possible natural increment in component size or a natural increment in the number of components. Option - (B) actively promotes a component size increment whilst option – (C) actively promotes a component count increment. This model suggests that where complexity is limited in one aspect of an application a corresponding growth in complexity is experienced elsewhere. Managing complexity in an application can therefore be viewed as a process of mediation between competing requirements.

Initial reaction to adverse complexity metrics should be to validate the systems architecture against stated objectives. If the architecture and objectives are themselves undergoing review the metrics should be judged in such a manner as to acknowledge future and past objectives.

Within the context of this application we can see that no one complexity management strategy was adopted. They utilised all techniques. They sought to control various aspects of the application through the consolidation of code. This was done via the promotion of the frequency of instances of indirect coupling and increasing the services which a class had to offer. To have sought to reduce the measures reflecting internal coupling - $CC_{I,}$ service provision - RFC and class functionality WMCM would therefore have been to act contrary to the re-design effort. In this restricted set of circumstance option – (C) was the appropriate choice.

An application can be dominated by the effect of a few unlimited and overly complex classes. The class count within this application increased as did the instance of child classes. It has been demonstrated that for some classes the actual class size was reduced as illustrated by the % decrease in the measure WMCLOC. Furthermore we know that classes were removed and replaced. The project team had therefore adopted a dual strategy and implemented refactoring of specific targeted classes option – (B).

Having implemented a re-structuring of the application, the project manager, thereafter accepted that certain classes would exhibit complexity as they were responsible for the intricate areas of functionality within the application. For this product and this Company, that decision was acceptable; the organisation structure facilitated a depth of product knowledge within the team that could cope with complex components. The active management of change within the application exhibited in the redesign effort allowed the product to stabilise and the team to take advantage of a controlled maintenance overhead. For a period of time the project team could reap the rewards of re-structuring and option – (A) became a viable alternative.

4. Conclusions

We began this paper by asking three questions concerning the appropriate course of action to take on observing complexity within an application. We have presented observations based upon research knowledge, commercial experience and an empirical investigation of a commercial application. The Company responsible for the application had actively attempted to redesign the system in order to promote robustness, flexibility and maintainability. Understandably this activity impacted upon the identifiably complex classes and sought to improve upon system architecture. The Company was satisfied that the outcomes of the second release of the application had met the re-design objectives.

We found that the majority of classes had experienced a reduction in complexity ranking. This however was not often a sufficiently significant reduction and therefore failed to remove the classes from being categorised as the most complex within the system. We also found that where complex classes had been removed, they were frequently replaced by classes of equal or greater complexity. Overall version 2 of the application exhibited an increase in the majority of measures. The only measure to decrease was WMCLOC which related to lines of code.

We have presented an analysis of the options available to a project manager when attempting to control complexity. Essentially these options include a deliberate acceptance of the current status of the application, the active management of complex classes by seeking to reduce the complexity of each component via sub-division and the active management of complexity by the promotion of a number of very complex classes. It was found that for this product re-structuring effort utilised each alternative. Our study suggests that when attempting to limit complexity results can be effective within a targeted component however the reduction of complexity in one aspect of an application often implies an increase elsewhere.

In answer to the above questions we would have to suggest that

1. A manager must give careful consideration to the maintenance cycles of an application and the reported complexity levels. No one answer is correct; the decision making process on whether to reduce complexity levels is in reality a mediation process between many competing factors within the development context.
2. The aim is not always the reduction of complex measures and certainly this response to the receipt of complexity indicators should not be applied blindly. Over time however the suggestion is that complexity within an application should be actively managed.
3. A detailed examination of a complex commercial application suggests that it is not possible to reduce complexity without removing functionality. Certain aspects of complexity may be actively managed and even reduced however this is often at the cost of increased complexity elsewhere.

In summary software systems are inherently complex. Where efforts are made to actively manage complexity a migration affect may be observed as a reduction of measures in one aspect of an application implies an increase elsewhere. Further studies of the evolution of complexity patterns within business applications are required in order to better understand beneficial maintenance patterns.

5. Acknowledgments

This work was funded by the Centre for Software Process Technologies at the University of Ulster which is supported by the EU Programme for Peace and Reconciliation in Northern Ireland and The Border Region of Ireland (PEACE II).

The authors would like to thank Meridio for their assistance in supplying the data set for this study.

6. References

[1] V. R., Basili, L. C. Briand, W. L. Melo, ‚*A Validation of Object Oriented Design Metrics as Quality Indicators*', IEEE Transactions on Software Engineering, Vol. 22, No. 10, 1996, pp. 751-761.

[2] B. Boehm, V.R..Basili, *'Top 10 List (Software Development)'*, Computer ,Volume: 34 , Issue: 1 , Jan. 2001 Pp:135 – 137.

[3] G. Booch, *'Object-Oriented Analysis and Design With Applications – Second Edition'*, The Benjamin/Cummings Publishing Company Inc., California, 1994.

[4] M. Cartwright, M. Shepperd, *An Empirical Investigation of an Object Oriented Software System*', IEEE Transactions on Software Engineering, Vol 20, No 8, August 2000, pp786-796.

[5] S. R. Chidamber, C. F. Kemerer, 'A Metrics Suite for Object Oriented Design.' IEEE Transactions on Software Engineering, Vol. 20 1994, pp 476-493.

[6] S. R. Chidamber, D. P. Darcy, C. F. Kermerer, '*Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis*', IEEE Transactions on Software Engineering, Vol. 24, August 1998 pp. 629-639.

[7] Harmer, T.J., Wilkie, F.G., 'An extensible metrics extraction environment for object-oriented programming languages,' Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation, 2002., pp:26 – 35.

[8] R. Subramanyam, M. S. Krishnan, *Empirical Analysis of CK Metrics for Object Oriented Design Complexity: Implications for Software Defects*', IEEE Transactions on Software Engineering, Vol 29, No 4, April 2003, pp297-310.

[9] F. G. Wilkie & T. J. Harmer, 'Tool Support for Measuring Complexity in Heterogeneous Object Oriented Software', IEEE Proceedings of the International Conference on Software Maintenance', 2002 p152 -161.

[10] F. G. Wilkie, B. Hylands, 'Measuring Complexity in C++ Application Software,' Software Practice and Experience, Vol. 28, No 5, April 1998, pp513-546.

[11] F. G. Wilkie, M. P. Ware, B. Kitchenham, T. J. Harmer, 'Evaluating the Sensitivity of Coupling Metrics to Evolving Software Systems', 'Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress', November 2004, pp 77-88.

Clone Evolution in Open Source Software Systems: When, How and Why do Software Maintainers Remove Clones

Filip Van Rysselberghe and Serge Demeyer
Lab On Re-Engineering, University Of Antwerp
filip.vanrysselberghe@ua.ac.be

Abstract

Cloning code is widely recognized as a threat to the long term maintainability of a software system, hence the "state-of-the-art" dictates to regularly remove the clones inside the code. Unfortunately, little is known about the "state-of-the-practice" with respect to the removal of clones, and consequently we do not know how clone removal fits into the overall software maintenance process. In this paper, we report on a study of the evolution of clones within four open source systems to see whether clones really get removed, and if they do – when, how and why it is done. Essentially, we have observed that clone removal differs considerably across the systems under study and suppose that this variety is due to the fact that clone removal is rarely an explicit part of the software maintenance process, but rather a side-effect of other (re)design activities.

1 Introduction

Code cloning or the act of copying a code fragment and making minor modifications, is generally considered an harmful practice since it hinders the maintainability of a system [16, 5]. As a result, a large number of researchers invested time and effort in the development of clone detection techniques and tools [2, 4, 6, 8, 9, 11, 12, 13, 15].

Despite the availability of these clone detection tools, little is known about the actual clone removal process. Do maintainers remove clones on a regular basis, or do they just eliminate the ones that hinder the maintenance activities? Should all clones be considered harmful, or is some degree of cloning inevitable, even beneficial? And if clones are removed, do maintainers remove the oldest ones first or the youngest ones? Answering such questions is necessary to gain deeper insight into viable clone removal strategies, yet only few authors have made an attempt in this direction. For instance, Antoniol et. al. discovered that the clone ratio of a system remains stable over time, suggesting that not all clones should be removed [1]. Also, Laguë et al. observe that many clones are removed over time, while another large set of clones never changes after their introduction, suggesting that the oldest clones have a tendency to remain [14]. In this paper, we try to verify and refine these initial observations to see whether we can deduce realistic clone removal strategies.

Therefore, we analyze the release history of four open source projects with varying characteristics (see table 1 for an overview) to see how software maintainers deal with clones in realistic circumstances. More precisely, we addressed following research questions:

- **Clone size**: *Do developers focus on one specific clone size, either large or small, or do they distribute their attention over all sizes of clones ?*
  Selecting the most urgent entities for removal is a recurring problem for software maintainers. Therefore this question evaluates whether size is a suitable indicator for priority. Based on the answer, future maintainers know whether it is good practice to target large clones first, or should rather remove all clones which hinder their current maintenance task, independent of that clone's size.
- **Clone location**:- *Do developers remove nearby clones (e.g. in the same file) as well as far away clones (e.g. in separate modules), or do they focus on only one of these categories?*
  Scalability is a major concern for clone detection tools. However the answer to this question may result in a reduction of the search space, e.g. when it shows that developers primarily target clones which are part of the same file. Furthermore, current evidence indicates that the distance between cloned fragments influences its removal [10].
- **Evolution**: *How does the number of clones evolve over time ?*
  Clone evolution studies have shown that the clone ratio remains stable over time, indicating that the number of clones increases in the same way as the number of procedures [1, 14]. This suggest that some degree of cloning is inevitable. Goal of this question is to verify

whether these observations generalize to other systems as well.

- **When**: *Do maintainers remove clones on a regular basis or is it part of a one-time reengineering effort* ?
  Clone removal requires a good understanding of the system's design and is effort intensive. Good understanding of when clone removal activities should get planned is thus required to be cost effective. Therefore this question studies how removal effort is distributed over time.
- **How**: *Which kind of restructuring (refactoring) techniques are used ?*
  Little is known about how to apply or combine refactorings (see [7]) to actually remove clones in a system [4, 3]. Based on the changes applied in the past we can leverage our knowledge on how to use restructuring techniques as refactoring.
- **Why**: *What is the motivation for removing clones ?*
  Not all clones are a severe threat for the maintainability of the system. Therefore this question helps us to discriminate negative from neutral (or even positive) clones. This knowledge can be applied to more accurately identify the situations in which the maintainability of the system is threatened.

2 Evolution Framework

Studying the evolution of clones through time requires a research framework which acquires all the necessary data and turns it into interpretation ready data. Therefore we developed a framework by the name DEVOL (Duplication EVOLution). DEVOL consists of three components namely a data acquisition component, a clone detection component and a report generation component.

The data acquisition component of DEVOL queries a versioning system, extracting all public releases. A set of public releases, available for further processing, is the result of the data acquisition component.

Clone detection is carried out by the tool CCFinder[9]. The result of the detection component consist of one clone report for each release extracted during data acquisition.

Afterwards these clone reports are summarized in one representation which abstracts from the unnecessary cloning details. To create such an abstraction, the number of clone relations per file of a release is counted. Afterwards the various counts for a file are placed in their chronological order. Figure 2 shows an example.

Based on the evolution reports produced by our framework, it is possible to draw some general conclusions about the evolution of clones. Furthermore the reports allow the identification of clone removals since those are reflected in the decrease of the number of cloning relations for a file. The details of the underlying clones can be found in the original clone reports.

	R1			R2			R3			R4			R5		
CookieTools.java	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0
Ajp12ConnectionHandler.java	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0
LocaleToCharsetMap.java	2	0	0	2	0	0	2	0	0	2	45	0	2	45	0
...															
ErrorHandler.java	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TOTAL	4	1	0	4	1	0	4	1	0	4	45	0	4	45	0

Figure 1. Example of a clone evolution summary for releases R1 to R6 of a project. Each column for a release like R4, corresponds to the number of cloning relationships for one category (in same file = 1, in same directory = 2 and other = 3). Coloring is used to highlight clone removals (gray) and additions (black).

3 Results

3.1. Clone size

To answer this question the number of clones which are removed from one release to another are counted for three size categories: small (between 50 and 100 tokens: ±10 to 20 LOC), medium (100 to 200 tokens: ± 20 to 40 LOC) and large (> 200 tokens). Afterwards the relative number of removals is calculated by dividing each absolute count by the number of clones before the removal.

The relative numbers for the different size categories are very similar in the projects Gaim and Tomcat. For Tomcat and ArgoUML on the other hand, we observe large differences. In these two last projects, the removal effort seems to focus on clones of 100 to 200 tokens (figure 2). Occasionally, the effort shifts to one of the two other size categories.

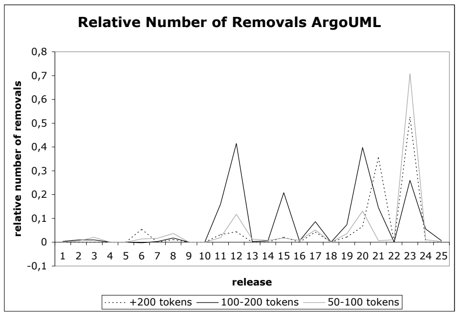


Figure 2. Graph displaying the evolution of the relative number of removals in ArgoUML for each category of clone sizes. It shows that primarily clones with a length between 100 and 200 tokens are targeted.

Project	Description	Language	Changes	Releases
ArgoUML	UML modeling tool with reengineering functionality	Java	User demands changes	26
Gaim	Multi-protocol instant messaging client	C (GTK)	protocol changes	39
JBoss	J2EE compliant application server	Java	performance and security changes	30
Jakarta-Tomcat	Servlet and Java Server Pages engine	Java	re-engineering changes	15

Table 1. Overview of the cases

3.2. Clone location

A similar approach as for the first research question can be adopted to answer this one. Here however, we have to compare the relative number of removals for three different categories namely intra-clones (clones within the same file), intra-package-clones (clones between files within the same directory) and inter-clones (clones between two unrelated system parts).

Comparison of the relative number of removals for each project shows an observable difference in the removal of clones based on their location. Clones which are in unrelated system parts for example are less frequently removed.

3.3. Evolution

We compared the number of clones over time with the actual code size evolution (in lines of code — LOC). Based on this comparison we argue that it is impossible to define a common clone evolution tendency.

Despite the fact that the code size increases over time for most projects, there is no common trend in the evolution of the number of clones. On one hand there are projects like Gaim and Tomcat for which the number of clones increases over time. While on the other hand, there is a project like ArgoUML for which the number of clones decreases. JBoss lies in between both tendencies since the number of clones stays more or less at the same level in this project.

3.4. When

For this fourth research question we compared the number of removals and additions over time. This way we observed that all projects undergo decrease as well as increase phases which indicates that clone control is not a continuous task. Additionally, we noticed that the developers of ArgoUML more frequently spend time to remove clones: maximum 2 increases exist in between two decreasing releases, while the maximum for Gaim is 5.

3.5. How

Manual classification of all code changes which influenced the existence of a clone shows that the majority of clone removals are not caused by refactoring changes. Instead, cloned fragments are regularly altered by simple changes as adding a statement or introducing a new condition. Besides these accidental changes, we observe a large number of changes to remove a code fragment which is no longer necessary. Figure 3 illustrates this dominance by showing the distribution of ArgoUML's changes.

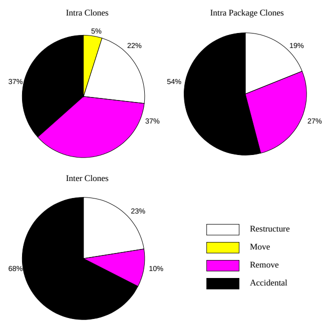


Figure 3. Distribution of change types for ArgoUML

A small number of changes of ArgoUML are classified as move changes. Although their number is quite small for ArgoUML, they are much more common in JBoss and Tomcat. Moving functionality around is very suitable for a good distribution of responsibilities in the system. For the reduction of the total number of clones it is unsuitable since it moves a clone from one file to another.

About 70% of ArgoUML's restructure changes can be classified as refactorings. Clones within the same file are usually removed by applying the "*extract method*"- refactoring. Many "instance of"-conditions are for example extracted into a separate method. The *"Extract superclass"*-, *"pull up method"*-, *"form template method"*- and *"encapsulate field"*-refactorings are applied to remove the two remaining clone types.

The remaining 30% of restructure changes is dedicated to more in-depth changes like the introduction of external resource files which removes data (initialization) clones. For Tomcat and JBoss, this percentage is much higher.

3.6. Why

Manual analysis of the log messages associated with the clone removing changes indicates that clone removal was almost never the goal. Instead the prime causes given, are the removal of deprecated code and design improvement. Although the latter could very well include clone removal, other reasons are mentioned in the messages. Sentences like ”moved to separate class”, ”centralize code” and ”only diagram specific in”, indicate that developers prefer coupling and cohesion as indicators of problematic design.

4 Conclusion

Our study of the evolution of clones in four open source systems did not produce an observation that readily generalizes over all four cases. Instead we noticed that the amount of clones in most cases grows over time. Especially Gaim and Tomcat support previous observations that the cloning ratio in a system remains stable [1, 14].

Furthermore, we observed that exactly those two cases remove clones infrequently and apply few structural changes. The other two cases, which applied clone removing changes more frequently, were able to maintain the amount of clones at a more or less stable level (JBoss) or even reduce it (ArgoUML). This indicates that frequent restructure changes are better suited for controlling the amount of clones in a system.

The JBoss and ArgoUML developers primarily focus their attention on the removal of medium sized clones, while the developers of Tomcat and Gaim distribute their attention over all possible clones, independent of their size. However all cases agree on the fact that the vicinity of cloned fragments has an impact on the removal.

Concerning the removal of clones, we noticed that most clones were removed because of accidental changes and code removals. It indicates that clone removal is not an explicit part of the software maintenance process. This claim is supported by our observation that most clone-removing-changes mention other reasons to apply the change than pure clone removal. However this also means that clones are good indicators for such problematic situations.

References

[1] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the Linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995. Received IEEE Outstanding Paper Award.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2000.

[4] I. Baxter, A. Yahin, L. Moura, and M. S. Anna. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–378, 1998.

[5] E. Burd and M. Munro. Investigating the maintenance implications of the replication of code. In *Proceedings of the International Conference on Software Maintenance*, page 322. IEEE Computer Society, 1997.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*. Computer Society Press, 1999.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126, September 1994.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654 – 670, 2002.

[10] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 85–94, September 2004.

[11] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[12] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[13] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301. IEEE Computer Society, 2001.

[14] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, pages 314–321. IEEE Computer Society, 1997.

[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254. Computer Society Press, 1996.

[16] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, pages 87–. IEEE Computer Society, 2002.

Quantitative Risk Assessment for Software Maintenance with Bayesian Principles

Mehmet Sahinoglu
Department of Computer Science, Troy University, Montgomery, AL
mesa@troy.edu

Abstract

Quantitative risk figures are needed not only to objectively compare alternatives and quantify monetary measures to budget for reducing or minimizing the existing risk [5]. They are also needed what particular vulnerability aspect or weakness of the software needs more care for maintenance after the fact. Among those existing analyses that favored a quantitative study, either i) there was no probabilistic frame about whether to add or multiply risks in a correct probabilistic frame of mind, or ii) the risk calculations were handled on one-to-one basis loosely in a non-systemic approach [1-3].

Therefore the maintenance priorities can be assessed by taking advantage of the security meter technique combined with the Bayesian procedures. Finally, some examples are cited from a simple hypothetical application since the simpler the example is, and the easier it is to comprehend the philosophy behind the maintenance-priority problem.

Key Words: *Bayesian, Security, Vulnerability, Threat, Maintenance, Priority, Countermeasure*

1. Motivation: Link between Software Risk and Maintenance

Software maintenance is the general process of changing a system after it has been delivered [7]. This strategy does not generally involve major architectural modifications. Software maintenance can imply, a) to repair software faults, where coding errors are the cheapest, and the design errors are more expensive and finally the requirement errors are the most expensive; b) to adapt the software to a new operating environment and c) to add or modify the system's functionality due to internal and external factors such as changing laws and markets or business structures. The proposed method will address the first two items, a) and b) for corrective and adaptive action by providing a quantitatively comparative risk assessment technique. It is a known fact of life now that software maintenance consumes 60-80% of most companies' software budgets, the largest single item contributed to high software costs [6]. Moreover growth in system size averages 10% per year [9] and maintenance expenditures generally increase as systems age. Therefore research efforts to integrate design and maintenance management policies to reduce unanticipated side effects have begun. Contrary to perfective maintenance, corrective maintenance identifies and corrects software performance and application failures, whereas adaptive maintenance conforms the software to new data requirements or processing environments in order to minimize functionality risk that arises when the environment changes. Traditionally maintenance cost does not measure future expected loss due to failures, only the historical cost of fixing the software. That is, the subjective judgments should not be alone but supported by quantitatively objective risk assessments to determine not only the proper type of maintenance, but where efforts should be focused [8]. However, until present, there has been little theoretical support for these assessments. The security meter approach can provide a quantitative comparison, and inform the analyst of a budgetary portfolio paving the way to repairing, maintaining or replacing that module to determine the most cost-effective maintenance strategy.

2. Introduction of the Quantitative Security Risk Model

Corrective risk management is defined as the total process of identifying, measuring, and minimizing uncertain events affecting resources. This definition also implies the process of bringing management such as a remedial action, and control into the risk analysis.

Basic ingredients of risk assessment are the vulnerability (weakness), and the threat to activate the vulnerability and the countermeasure, which is an action, or device, or procedure, technique, or other measure that reduces the risk to an information system.

The proposed physical model identifies the deterministic constants and probabilistic inputs for the target output of the residual risk and cost to mitigate the risk [4].

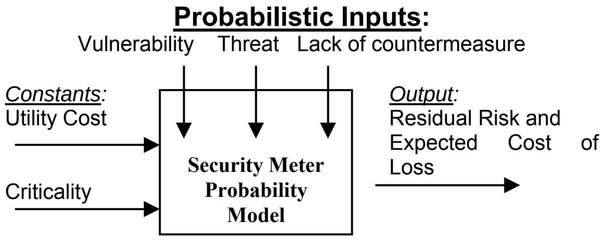


Figure 1. Security Meter Probability Model

Probabilistic Inputs: Vulnerability risk values range from 0.0 to 1.0, but the vulnerability values should add up to one. Each vulnerability has from one to several threats. Threat is defined as the probability of the exploitation of some vulnerability or weakness within a time frame under the conditions encountered. Threats with a range from 0.0 to 1.0, which are assigned to each vulnerability, also should add up to one. These undesirable threats taking advantage of the hardware and software vulnerabilities can impact in breach of the overall security umbrella. Each "threat" has a "countermeasure," whose complement gives the "lack of countermeasure." Both countermeasure and lack of countermeasure risk values (between 0 and 1) for each threat should add up to one.

Deterministic Inputs: System criticality from 0.0 to 1.0 indicates the degree of how critical or disruptive the system is in the event of entire loss. Criticality is zero if the residual risk is of no significance, like the malfunctioning of a printer, which has a backup. However, it is unity when like a nuclear power plant, its security is one of life and death with vital security concerns to humans. Utility cost is the total loss in currency units such as USD to the user for the particular system if the system is lost entirely.

Decision-Tree Diagram: Given that in a simple sample scenario, there are two or three or more of each choice, the following probabilistic frame holds. Sum of V_i=1 and sum of T_{ij}=1 for each i, and sum of LCM_k+CM_k=1 for each j, within a tree diagram as follows:

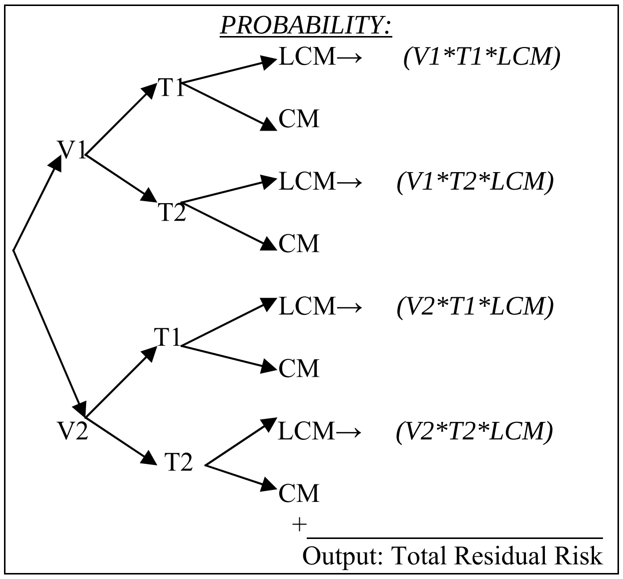


Figure 2. A Simplest Tree Diagram for Two Threats per each of the Two Vulnerabilities

Output: From the inputs the output Residual Risk is calculated, here,

*Residual Risk = Vulnerability * Threat * Lack of Countermeasure* (1)

To this calculated Residual Risk, the criticality and utility cost is applied to determine the Cost to mitigate the entirety of the vulnerabilities, where,

*Expected Cost of Loss = Residual Risk * Criticality * Utility Cost* (2)

An example, and its Monte Carlo simulated results are shown in a JAVA Applet tabloids in the Appendix.

3. A Security Meter Model Application

Risk analysis has various inputs like types of vulnerability, and threat and countermeasure for each threat. Criticality and the utility cost are constants as well the number of simulation runs. From these input values, we determine the output cost to mitigate the residual risk is determined. The data and output are given below:

Vulnerability	Threat	Countermeasure
Software Failure (Chance)	1. Design & Coding Error 2. System Down	1. Pre-release Testing 2. In-house generator
Software Failure (Intentional)	1. Virus 2. Hacking	1.Install antivirus software 2. Install firewall

Table 1: Vulnerability-Threat-Countermeasure Spreadsheet for an Office Computer

0.5	0.5	0.5	
		0.5	0.125
	0.5	0.5	
		0.5	0.125
0.5	0.5	0.5	
		0.5	0.125
	0.5	0.5	
		0.5	0.125
		Summed Residual Risk:	**0.500**

Table 2: Input Data (Expected Values) and Calculated Risk for Table 1 and Figure 2

*Final Risk = Residual Risk * Criticality= 0.5 * 0.5 = 0.25, where Criticality = 0.5*
*Expected Cost = Utility Cost * Final Risk = \$1000 * 0.25= \$250, Utility Cost = \$1000*
Final Result: (Maximum) Expected Cost of Repair or Loss= \$250

4. Bayesian Applications for Software Maintenance

By using a single shot for one simulation trial, shown in Appendix 1, assuming that it is a hypothetical example, let's apply Bayesian to determine the vulnerability to need the most care for maintenance.

Let's now ask the Bayesian question in relevance to our maintenance problem. Given that the office computer software is R for risk (or failed due to chance or malicious causes), what is the probability that it is due to chance (design/system down), or malicious (virus/hacking) causes. Statistically, we need to find the Bayesian probabilities listed below:

P (Design Error |R) =0.097097/0.506371=0.1917 (3)

P (System Down |R)=0.118578/0.506371=0.2341 (4)

P (Virus Attack |R) =0.151268/0.506371=0.2987 (5)

P (Hacking Attack |R) =0.139429/0.506371=0.2755 (6)

From these Bayesian aposteriori probabilities, it is obvious to judge that the posterior risk (R) due to chance failures of the first vulnerability is .1917+.2341=.4258, or 42.58%. On the other hand, the prior contribution of the chance failures at the very beginning stage was more; .4645 or 46.45% in Table 3.

On the other hand, the prior contribution of vulnerability of the malicious failures was .5354 or 53.54%. The aposteriori contribution resulted to be 0.2987+0.2755=0.5742 or 57.42%. What this means is that although malicious causes of the second vulnerability constitute 53.54% of the totality of failures, these causes generate 57.42% of the risks. The implication is that more severe care for software maintenance is required on the second vulnerability than the first one, proportionately. For corrective maintenance at this stage, two remedial measures are feasible in the order of applicability:
1) Work to improve on the countermeasures for the vulnerability B, especially to note virus attacks constitute more than half (29.87% > 27.55%) in this hypothetical example.
2) Then after preventive or corrective measures on the vulnerability with the priority, rerun the sec-meter analysis to compute the updated Bayesian probabilities aposteriori if any improvement is recorded by comparing the expected costs of loss between the pre- and post maintenance.

5. Conclusions

The proposed security meter approach is a quick and a bird's-eye-view symptomatic way of calculating a component or system's software security risk. Some other techniques used hitherto such as "attack trees" don't provide an accurate overall picture [1-3]. The vulnerabilities that need more surveillance can be ranked from the worst to the best through a-posteriori Bayesian analysis. This is very useful for prioritization in the vast arena of software maintenance [6-9]. The proposed model is supported by a Monte Carlo Simulation that provides a purely quantitative alternative to those conventional qualitative models [4, 5] as in Appendix. One assumes that the vulnerability-threat-countermeasure input data will be available and reliable. The main concern in this paper is the security meter model proposed to assist a sound decision-making in the choice of maintenance priorities rather than the potential data concerns.

References

[1] E. Forni (CISSP), "Certification and Accreditation", DSD Laboratories, Virginia, 2002
[2] B. Schneier, "Attack Trees", Dr. Dobbs' Journal, December 1999
[3] Integrated Research in Risk Analysis and Decision Making in a Democratic Society, National Science Foundation, Workshop Report (92 pages), Arlington, VA, July 2002

[4] M. Sahinoglu, "Security Meter- A Probabilistic Framework to Quantify Security Risk", Certificate of Registration, US Copyright Office, Short Form TXu 1-134-116, December 05, 2003

[5] M. Sahinoglu, "Security Meter- A Practical Decision Tree Model to Quantify Risk," IEEE Security and Privacy Magazine, April/May 2005, pp.18-24

[6] S. A. Scherer, Software Failure Risk, Plenum Press, New York, 1992

[7] J. Keyes, Software Engineering Handbook, Auerbach Publications, CRC Press, 2003

[8] E. B. Swanson and C. M. Beath, "Departmentalization in Software Development and Maintenance," Comm. ACM 33(6), June 1990, pp.658-667

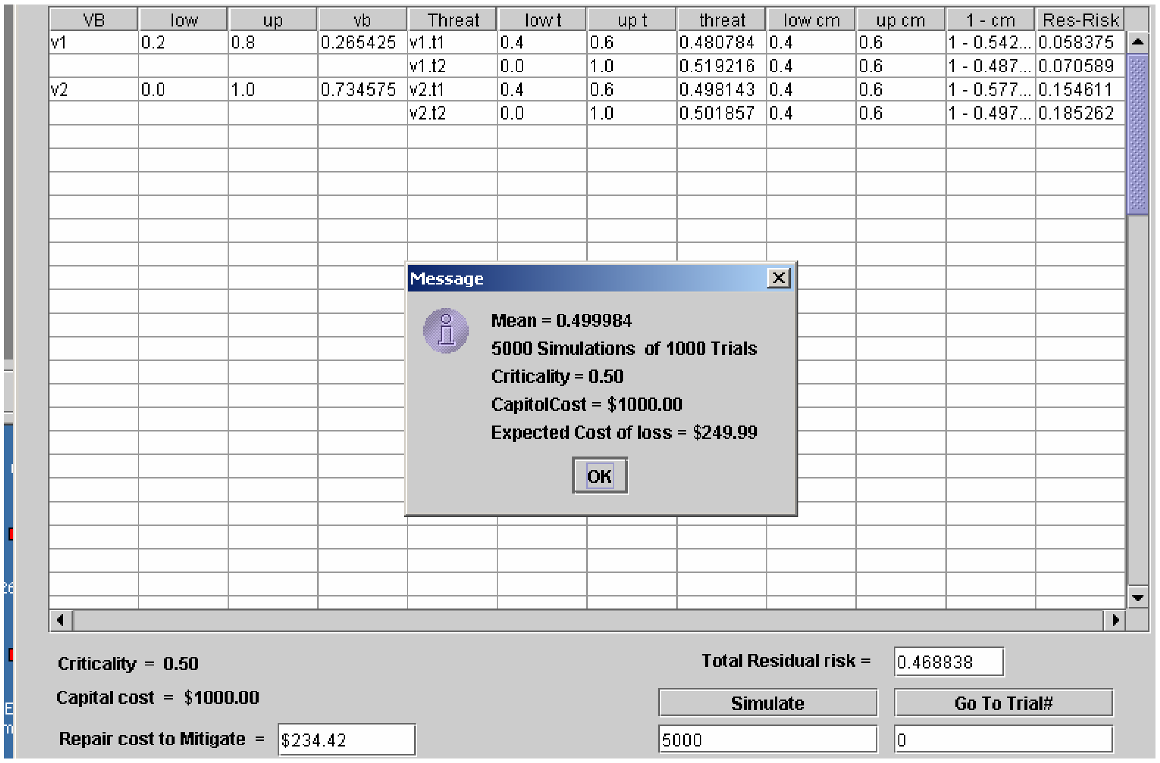
[9] G. Parikh, Handbook of Software Maintenance, New York: Wiley, 1986

[10] J. F. Freund, Mathematical Statistics, Prentice Hall, NJ, Fifth Edition, 1992

Appendix

Vulnerability	b	a	ran.value	Threat	b	a	ran.value	$_b\text{LCM}_a$		LCM	Risk
Chance Failure (A)	0.2	0.8	**0.464538**	Design Error	0.4	0.6	**0.496143**	0.4	0.6	**0.421288**	*0.097097*
				System Down	subtr	subtr	**0.503857**	0.4	0.6	**0.506611**	*0.118578*
Malicious Failure (B)	subtr	subtr	**0.535462**	Virus	0.4	0.6	**0.490508**	0.4	0.6	**0.575932**	*0.151268*
				Hacking	subtr	subtr	**0.509492**	0.4	0.6	**0.511076**	*0.139429*
										Residual Risk	*.506371*

Table 3: One simulation result in Table 2, assuming bold values are the real observations.



VB	low	up	vb	Threat	low t	up t	threat	low cm	up cm	1 - cm	Res-Risk
v1	0.2	0.8	0.265425	v1.t1	0.4	0.6	0.480784	0.4	0.6	1 - 0.542...	0.058375
				v1.t2	0.0	1.0	0.519216	0.4	0.6	1 - 0.487...	0.070589
v2	0.0	1.0	0.734575	v2.t1	0.4	0.6	0.498143	0.4	0.6	1 - 0.577...	0.154611
				v2.t2	0.0	1.0	0.501857	0.4	0.6	1 - 0.497...	0.185262

Message

Mean = 0.499984
5000 Simulations of 1000 Trials
Criticality = 0.50
CapitolCost = $1000.00
Expected Cost of loss = $249.99

OK

Criticality = 0.50
Capital cost = $1000.00
Repair cost to Mitigate = $234.42

Total Residual risk = 0.468838

Simulate 5000
Go To Trial# 0

Table 4: Monte Carlo Simulation results after 5 million runs for the example in Table 1 & 2.

DDL: Extending Dynamic Linking for Program Customization, Analysis, and Evolution

Sumant Tambe   Navin Vedagiri   Naoman Abbas   Jonathan E. Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

While new software languages and environments have moved towards richer introspective and manipulatable runtime environments, there is still much traditional software that is compiled into platform-specific executables and runs in a context that does not easily offer such luxuries. Yet even in these environments, mechanisms such as dynamic link libraries do offer the potential for more control over the deployed and running system, and can offer opportunities for supporting dynamic evolution of such systems. In this paper, we present our initial explorations into building such support. Our approach is to extend the Gnu open-source dynamic linker to give the deployer control over the configuration of the system, and to be able to dynamically evolve that system. Applications of this capability include runtime component configuration, program evolution and version management, and runtime monitoring.

1. Introduction

Dynamic link libraries, also called shared libraries or shared objects, have the potential to offer a rich, dynamic, component-based deployment platform. Many, if not most, of the latest ideas in component-based software frameworks and development could be supported by the shared object platform. Private namespaces, naming service lookup for binding requests, interface checking, introspection and monitoring support, and dynamic reconfiguration are just some of the capabilities that could be supported—if the underlying framework enabled them to be.

In this paper we describe DDL, a tool based on modifications and extensions to the dynamic linker that enable dynamic control over the linking process, and allow features such as those listed above to be implemented. DDL allows the user to control the runtime configuration of the application, enables the easy construction of runtime monitoring tools and supports the runtime evolution of dynamically linked programs. DDL is a modification of the Gnu dynamic linker, which is part of the Gnu C library. Our current tests have only been on the Gnu/Linux/ELF/x86 platform, although the Gnu libraries (and the dynamic linker) are ported to many other platforms.

Section 2 discusses dynamic linking from an architectural perspective. Section 3 details our modifications to dynamic linking that allow link interception and runtime evolution. Section 4 presents related work, and finally Section 5 concludes with some ideas for the future directions that we are pursuing.

This work was supported in part by the National Science Foundation under grants CCR-0306457 and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

2. An Architectural Perspective

Stepping back from the low-level idea of dynamic linking being symbol resolution, a broader picture of the meaning of what is happening appears. In using dynamic linking, an executable program is *incomplete*—it does not have the complete code to run. Instead, it needs external *services* to be able to run to completion. The dynamic linker's job is to find these services and connect the program to them so that it can use them. This view captures the idea that dynamic linking supports an architectural, component- and connector-based view of the application [11].

Each shared object, including the application program, is a component that contains references to required services. These references are in the form of names (symbols). Additionally, each component also advertises its provided services, also referring to these by using names (symbols). There is no reason at all why the required service name must match the provided service name—indeed, it is constraints like these that cause global namespace pollution.

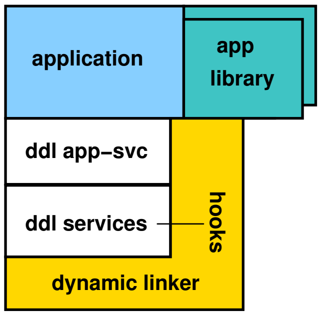


Figure 1. DDL system architecture.

Rather, we can view the names as local specifiers of required and provided services. It is the architectural level that needs to provide a binding specification between component namespaces. This might be as simple as equating them (thus reducing the problem to normal linking), but it might involve translation of one name to another, binding a complex connector in between, binding to a remote service, or even more complex configuration processes.

Understanding dynamic linking as just one mechanism for connecting independent components into a complete application, one in which the default behavior of symbol matching is a throwback to a programming-language-centric and monolithic-system viewpoint, enables us to place it alongside modern component system frameworks and to work to bring its implementation—the dynamic linker—up to date in its capabilities.

3. DDL: A Dynamic Dynamic Linker

DDL is an extension to the Gnu dynamic linker, and is extensible itself. Figure 1 shows the high-level system architecture that DDL implements. The shaded portions indicate parts of the system that DDL does not modify. The application and application libraries are not modified, at the source or binary level, and the bulk of the system dynamic linker is unmodified.

We have been careful to make DDL thread-safe. The regular dynamic linker is by default thread safe because it always updates its global data structures equivalently, so it does not matter if threads interleave their updates. With DDL this may not always be the case, and DDL needs some of its own static data in any case. DDL does support both threaded applications, and client tools that create their own threads. We did not, however, make DDL depend on a thread library. We did not want to add any library dependencies that the application does not already use. Instead, since our initial tests indicate that overlapping link requests from multiple threads are vanishingly rare, we use simple busy waiting in very small sections of code rather than *pthread*-based library routines.

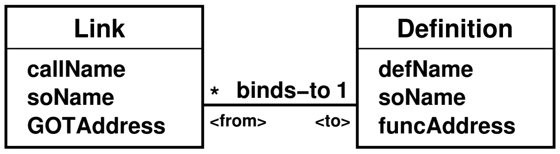


Figure 2. Link and definition UML.

One goal in this project was to have very minimal modification to the Gnu dynamic linker itself. We decided that any significant code we developed would sit outside of the linker. Thus, our essential modifications boil down to callback hooks in the linker code itself. In this, the hooks built into the dynamic linker do not *provide* an API to external services but rather they *use* an API provided by the DDL control library. The DDL control library and the tools that use it are thus event-driven and passive, unless they spawn their own threads.

3.1. Link Interception and Redirection

The fundamental capability that DDL supports is link interception and redirection. This allows DDL and the tools that use it to monitor and control the dynamic linking process. Each time a link is being resolved by symbol lookup, the callback hooks invoke user code for several purposes: to allow symbol modification to redirect the binding to a different symbol; to inform the tool of the actual symbol that was looked up, its resolved address, and the address of the link entry; and to allow for an offset to be added to the resolved address.

Thus, with this interface, tools using DDL can inspect every link request, choose whether to redirect it to another symbol name, and record the information about the resolved symbol and about the link itself. By removing the implicit equality between requested and provided symbols in the linking operation, we reify these two notions into separate concepts, as shown in Figure 2, which shows how the *links* and *definitions* relate to each other and the information that uniquely describes each of them.

A link is defined by the original symbol name it is supposed to be linked to, the name of the shared object it is for (the main application or a shared library), and the address of the GOT entry where its binding address is stored.[1] A definition is defined by its symbol name, the name of the shared object it exists in, and the address at which it exists. Further information that is useful to save is a reference

1 In ELF dynamic linking, each shared object has a *global offset table*, or GOT, where addresses of externally defined symbols are entered by the dynamic linker. Code that references these symbols use the GOT entries.

from the link to its current definition, and in the reverse direction a set of links that currently reference a given definition. We implement this at the DDL services layer, so that tool builders do not need to re-implement it.

Without using our redirection capabilities, all links will point to definitions of the same name. There are potentially multiple links to the same definition because each shared object that calls that function will have its own link (i.e., its own GOT and its own unique GOT entry for calls to that function). If our redirection capabilities are being used, then the called name associated with the link can be different from the defined name of the definition; thus it is important to keep track of these names separately.

Since called functions cannot in general differentiate between their invocations, it does not make sense to redirect multiple different function invocations to the same function (although DDL will allow this). However, there are cases in runtime monitoring and program maintenance where it would be useful to have a *concentrator* function that did receive calls for multiple symbols and was able to differentiate them. For example, if one wanted to trace all calls in a program, it would be nice to have a single wrapper do the job rather than a unique wrapper for every function.

To support this functionality, we utilize an offset-based mechanism, where an offset to a symbol value provides a direct path to a unique *jump table* entry, which will set an ID for the call and then jump to the actual concentrator function. To use this capability, we first must create a jump table. A simple (non-thread-safe) example of this is below.

```
unsigned int func_id;

void wrapper_jmp()
{
   asm("   movl $0, func_id
           jmp wrapper
           movl $1, func_id
           jmp wrapper
           ...
           movl $99, func_id
           jmp wrapper");
}
```

This example represents a 100-entry jump table, where each entry is a move/jump instruction pair that sets a global variable to its index value and then jumps to the wrapper function. Note that the program never *calls* the *wrapper_jmp* function—rather, it calls directly to one of the entries, which in turn jumps to the wrapper. Our use of DDL would redirect each symbol to $(wrapper\_jmp + 3 + i * 15)$, where i is the entry assigned to that symbol. To do this it would do symbol redirection to "wrapper_jmp", allow the dynamic linker to find that symbol, and then add the offset using the DDL offset callback. At the same time, in our DDL extension we would save the symbol string in a string table, at the same index being used in the jump table.

The wrapper function, using the global $func\_id$, would have access to the index of the function currently being called, and from there could get the name of the function (since we saved it). After doing its tracing behavior (or whatever it is supposed to do), it could use *dlsym()* to resolve the original function and to call it. Functions with different argument vector lengths can still be handled by the same wrapper, since the reverse-calling convention ensures that extra argument data is ignored. The wrapper only needs to know the maximum argument bytes it needs to push on the stack.

While the jump table must be created in a platform-dependent manner, the basic idea remains essentially the same on most platforms, and through the symbol-plus-offset mechanism that DDL exposes to the user, effective use of a single site for multiple redirected symbols can be accomplished.

As one example, we used this capability to fully trace the SimpleScalar CPU simulator [1].

3.2. Runtime Link Modification

DDL provides all the information needed to maintain an internal data structure of definitions and links: resolved symbols and their addresses, and addresses of GOT entries and the current definition of the function a link is referring to. Maintaining this information during the runtime of the program allows us to support dynamic program evolution through runtime link modification.

In order to modify a link, we simply need to change the address that is in its GOT entry to be the address of some other function. All the subsequent calls through that link will be directed to the new function. Note that these calls are from all the call sites in the shared object whose link we just modified. Thus, the granularity of program evolution is at the shared object level. This can be used along with runtime loading of new shared objects (e.g., using *dlopen()*) to update a running system with new functionality.

Other work in dynamic program evolution has noted a desire to perform transactional updating—making sure that a module is not being actively used before updating references to or away from it [12]. This often boils down to checking the call stack to see if any functions in the module are active. We have not yet concerned ourselves with providing such capability, but plan to investigate these aspects in the future. In our current mechanism, existing calls through links being modified have already invoked the old definition, and those will eventually complete.

4. Related Work

The DITools project [13] is the closest related work to our DDL project. They used a similar approach to link interception and modification, and supported redirecting a link to a wrapper and also an event notification mechanism where each monitored call was not wrapped but did generate an event to a fixed-interface callback. It does not appear that they addressed the issues surrounding C++, nor did they do non-function symbol resolution nor runtime link modification.

Ho and Olsson [8] describe *dld*, a tool for "genuine" dynamic linking. Their tool provides the capability to load and unload shared libraries, breaking links when a library is unloaded and relinking them to new code when new libraries are loaded. However, it does not appear that they ever supported redirection of links to different symbol names.

Thain and Livny [14] developed Bypass, which intercepts both system calls and regular procedures, but it produces unique managers for each system description, while our work is generic. Both the UFO [2] and SLIC [6] systems focus on system call interception and not generic application management. It appears that all of these only support static recomposition of an application, not dynamic modification at runtime.

Hicks et. al [7] work on binary software updating from a formal perspective. Their methods use typed, proof-carrying assembly code from which they can verify that an update will be safe. Their infrastructure includes special languages and compilers to generate the annotated assembly code, and a runtime framework that uses it.

Additional systems that provide instrumentation capabilities on executable binaries exist. Dyninst [3] can patch custom code into pre-existing executable code, and has provided a platform for several research tools. Detours [9] also does binary rewriting similar to Dyninst.

There is much work in dynamic introspection and modification of Java programs, but since this work is in a very different environment than ours, we do not explain it in detail here. Some representative references are [4, 5, 10, 12].

5. Conclusion

We have presented DDL, an extension to the standard dynamic linker that allows introspection and modification of the dynamic linking process. This capability supports a wide range of uses for software engineering practitioners and researchers, including a foundation for runtime monitoring and dynamic analyses, dynamic runtime program evolution, and other ideas that can use control over the linking process.

In our future work we are pursuing the use of this platform to support multi-version software fault tolerance and evolution, and to support our ongoing research interests in dynamic analysis. We are building an event-based extensible tool framework on top of DDL, and we have an initial prototype of reliable on-line C++ class evolution through the execution of multiple versions of classes. We hope that other researchers and practitioners will find DDL useful, and we will be working towards getting some form of our changes regularized back into the Gnu dynamic linker project. Our project can be found at http://www.cs.nmsu.edu/please/ddl/index.php.

References

[1] N. Abbas, S. Tambe, R. Srinivasan, and J. Cook. Using DDL to understand and modify SimpleScalar. In *Proc. 2004 Working Conference on Reverse Engineering*, page to appear, Oct. 2004.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A Personal Global File System Based on User-Level Extensions to the Operating System. In *ACM Transactions on Computer Systems*, pages 207–233, Aug. 1998.

[3] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000. www.dyninst.org.

[4] M. Dahm. Byte Code Engineering Library. 2002. http://jakarta.apache.org/bcel/.

[5] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[6] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX Annual Technical Conference*, June 1998.

[7] M. Hicks, J. Moore, and S. Nettles. Dynamic Software Updating. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[8] W. Ho and R. Olsson. An Approach to Genuine Dynamic Linking. *Software Practice and Experience*, 21(4):375–390, 1991.

[9] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. Technical Report MSR- TR-98-33, Feb. 1999.

[10] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proc. European Conference on Object-Oriented Programming*, pages 337–361, 2000.

[11] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[12] A. Orso, A. Rao, and M. Harrold. A Technique for Dynamic Updating of Java Software. In *Proc. 2002 International Conference on Software Maintenance*, pages 649–658, Oct. 2002.

[13] A. Serra, N. Navarro, and T. Cortes. DITools: Application-level Support for Dynamic Extension and Flexible Composition. In *Proc. 2000 Usenix Technical Conference*, pages 225–238, June 2000.

[14] D. Thain and M. Livny. Bypass: A Tool for Building Split Execution Systems. In *Ninth IEEE International Symposium on High Performance Distributed Computing*, pages 79–85, Aug. 2000.

Archeology of Code Duplication : Recovering Duplication Chains From Small Duplication Fragments

Richard Wettel Radu Marinescu

LOOSE Research Group
Institute e-Austria Timişoara, Romania
{wettel,radum}@cs.utt.ro

Abstract

Code duplication is a common problem, and a well-known sign of bad design. As a result of that, in the last decade, the issue of detecting code duplication led to various solutions and tools that can automatically find duplicated blocks of code. However, duplicated fragments rarely remain identical after they are copied; they are oftentimes modified here and there. This adaptation usually "scatters" the duplicated code block into a large amount of small "islands" of duplication, which detected and analyzed separately hide the real magnitude and impact of the duplicated block. In this paper we propose a novel, automated approach for recovering duplication blocks, by composing small isolated fragments of duplication into larger and more relevant duplication chains. We validate both the efficiency and the scalability of the approach by applying it on several well known open-source case-studies and discussing some relevant findings. By recovering such duplication chains, the maintenance engineer is provided with additional cases of duplication that can lead to relevant refactorings, and which are usually missed by other detection methods.

Keywords: code duplication, design flaws, quality assurance

1 Introduction

Duplicating code, while easy and cheap during the development phase, moves the burden towards the already overloaded and much more expensive maintenance phase. Fowler and Beck ranks it first in their list of "bad smells in code" [6] and we strongly believe they were right. Therefore, we will not emphasize the consequences of introducing duplicated code.

In a line-based approach, large blocks of code affected by modifications (renaming of variables or even statement insertions or removals) would be identified as small, less important fragments of duplicated code, apparently not related to each other.

To address this issue, we propose an approach that can merge such small fragments that belong together and provides the maintainer with some additional duplication blocks otherwise granted with less importance.

2 The Archeology Metaphor

Like an archeologist who puts together all the ruins of an ancient village in order to build a complete picture, rather than analyzing each artifact separately, we try to recover a close representation of a scattered duplicated block, in order to make the right refactoring decisions.

2.1 It Started with a Scatter-Plot

The scatter-plot approach, successfully applied in the code duplication detection field starting with the early '90 ([1], [4], [5]), uses a visual representation that can point out "dark" areas, which possibly host problems. Our approach provides the reengineer with results in form of a list of duplication chains. However, since the visual idea of a scatter-plot is behind our detection algorithm, we chose it as a means to illustrate the various concepts throughout this paper.

2.2 Need for Duplication Chains

Imagine we have the two pieces of Java code from Figure 1. Despite the fact that it seems obvious that they have common origins, due to the deleted and modified lines of code, they could be detected as 3 smaller clones, which is rather false. In a more pessimistic scenario, they would be filtered out by the minimum length threshold. One could rightly argue that there are approaches which can detect variables renaming. What if the lines of code are modified further than just variables or if there are lines appearing in only one of the two code fragments?

```
initSensors(tSensors);
readSensors(tSensors);
lcd.init();
int i = 0;
while(i < tSensors.length){
  temp[i] =tSensors[i].getTemp();
  lcd.println("T"+i+"="+temp[i]);
  i++;
}
regulateTemp(temp);
```

```
initSensors(tSensors);
readSensors(tSensors);
int i = 0;
while(i < tSensor.length){
  temp[i] =tSensor[i].getTemp();
  System.out.println("T"+i+"="+temp[i]);
  i++;
}
regulateTemp(temp);
```

Figure 1. Scattered duplication

Moreover, detected clones might not be relevant if they are too small or analyzed in isolation. Our main goal is to capture, along with the usual clones, blocks of scattered clones that may have common origin, which we will further refer to as **duplication chains**.

2.3 Anatomy of a Duplication Chain

A *duplication chain* can be a complex element (the representation of the recovered duplicated code block), composed of a number of smaller, exact clones (further referred as exact chunks), separated pairwise by non-matching gaps. Figure 2 illustrates the previous example's scatter-plot representation, where the marked cells correspond to the matching pair of lines of code intersecting in that precise point.

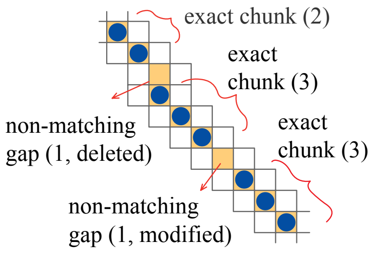


Figure 2. Duplication chain

An *exact chunk*, put in the context of the archeology metaphor, is a non-altered part of a duplicated block, that preserved its identity. An exact chunk appears in a scatter-plot as a continuous diagonal, as it can be seen on Figure 2.

A *non-matching gap* reflects the changes that have been made to the originating duplicated block, in terms of lines of code (insertion, deletion, modification). Thus, while apparently less important in clone detection, these non-matching parts provide us with extra information about the adaptation process. In a scatter-plot representation, non-matching gaps appear as shortest non-marked paths linking two consecutive diagonals (Figure 2).

A characteristic of a duplication chain, directly related to the adaptation process is the *signature*, which captures the structural configuration in terms of exact chunks, non-matching gaps and the metrics around them. In terms of the archeology metaphor, the signature could be associated with a "map" storing the places where all the related items where discovered. The signature of the previous example is "E2.D1.E3.M1.E3", which describes two code fragments having 3 exact (E) chunks of sizes 2, 3 and 3, separated by 2 non-matching gaps: one with 1 deleted (D) line and the other with 1 modified (M) line of code.

2.4 Proportional Harmony

In the context of size, we want to capture only those code fragments pairs that contain a *significant* amount of duplication. While an exact clone is significant if the clone's size is larger than a threshold, a significant duplication chain must also be proportionally harmonious. First, we will define some metrics related to these proportions (measured in LOC).

Size of Exact Chunk (SEC) reflects the degree of the granulation left behind by the adaptation phase of the copy-paste-adaptation process. *Line Bias* (LB) is the size of non-matching gaps and its value may allow us to decide if two exact chunks belong to the same duplicated block, since it provides a measure of distance between them. *Size of Duplication Chain* (SDC) is the size of the more meaningful block of duplication, which actually suggests its magnitude.

In order to constrain the duplication chain's proportions, we will set a minimum SDC to filter the less relevant clones. Furthermore, we will impose a minimum SEC and a maximum LB. In the harmony context, there is a relation between SEC and LB: the SEC should always be larger than LB, because it is not desirable to detect duplication chains with gaps larger than its exact chunks.

2.5 Stepwise Recovery Methodology

We propose an approach of lightweight line-matching, enhanced with the concept of chain duplication, which can also cover duplications that cannot be detected by a simple line-matching approach.

Phase 1: Code Preprocessing. After reading the source-files, we eliminate the white spaces, so that the various indentation styles would not make the difference. An optional feature is the possibility to ignore the comments in the analysis process. This phase provides a set of relevant (clean) lines of code.

Phase 2: Populate the scatter-plot. As in the original scatter-plot approach, we compare every line of code with every line of code in the project. As a result of this comparison, the matrix will be divided in two symmetric areas, around the main diagonal, which is always completely marked (self comparison). We then populate only one half of the matrix, in order to avoid storing redundant information. The matching intersections are marked.

Phase 3: Build the duplication chains. Starting with the left-upper matrix cell, we look for the first marked one, as a starting point for a possible duplication chain. From here, we accumulate the marked cells following the diagonal direction towards the lower-right cell. The algorithm will accept as an extension of the chain either a marked cell that continues an exact chunk or a marked cell situated in its vicinity, whose range is controlled by the maximum LB. Significant duplication chains will be stored in a results list.

3 Validating the Approach

In order to present the advantages over traditional code duplication detectors, we have to prove that this approach provides additional relevant duplications, usually missed by other line-based detection methods. To demonstrate this, we applied the proposed approach over a set of case studies. DuDe, our supporting tool owes its flexibility to the tunable thresholds which can filter the results based on size and proportional harmony. The tool provides a list of suspects which can be further analyzed, by means of the duplicated code visualization feature and statistical information.

3.1 Quantitative Gain

We took 8 Java and C projects, covering the size range from 0.5 MB to 10 MB, containing between 11,000 and 235,000 LOC in a number of 65 to 741 files. We compared traditional approach results (NODC1, COV1) with the one based on duplication chains (NODC2, COV2). Correlating the results in terms of *coverage*[1] and number of duplication chains presented in Figure 3, we can state that our enhanced "archeological" approach provides an important amount of otherwise lost code duplication information.

Project Name	Lang.	NOF	Size (MB)	KLOC	NODC 1	NODC 2	COV1 (%)	COV2(%)
weltab	C	65	0.43	11	759	711	72	76
cook	C	590	2.68	80	1285	1744	9	16
snns	C	420	4.82	115	47930	53274	16	21
postgresql	C	612	9.52	235	704	1070	8	11
netbeans-javadoc	Java	101	0.68	14	39	48	12	15
eclipse-ant	Java	178	1.43	35	14	24	2	4
eclipse-jdtcore	Java	741	6.9	148	716	1127	12	16
j2sdj1.4.0-javax-swing	Java	538	8.39	204	1171	1388	7	10

Figure 3. Experimental results

3.2 Quality-Focused Analysis

In order to validate the quality of our results, we extracted the clones found in another project (JHotDraw) only by the duplication chain approach and analyzed them manually. Out of 72 clones, there were 30 duplication chains. Summarized, we found over 76% relevant clones, potential subjects to refactorings.

In order to calculate the *recall*[2] of our tool, we considered only the type 1 (exact) clones from the reference set built in [3] belonging to the biggest project (eclipse-jdtcore) and intersected it with the duplication chains set found by DuDe. Concluding, our tool's recall was 89% under the strict conditions of the [3] experiment, but in a more loose context the recall could rise up to 95%.

3.3 Validation of Scalability

The largest project over which we successfully applied our approach, was a C project with 32 MB of source code and over 600,000 LOC. The analysis took 2h45m, which is an acceptable amount of time for such an industrial-size project.

4 Related Work

The idea of analyzing the non-matching parts of the duplication appeared in [10], where the authors used it to observe evolution between several versions of a system. An interesting contribution, similar to our approach was [11], whose authors address the gapped clones issue by combining the exact clones provided by their token-based clone detector [7]. Various other techniques for detecting clones have been proposed over the years: based on scatter-plots [1], [4] and [5], on metrics [9] or abstract syntax trees [2] and program dependency graphs [8].

[1]Coverage is the ratio of the number of copied lines of code to the total number of lines in the system

[2]Recall is the percentage of discovered clones over existing clones

5 Conclusions

5.1 Pros and Cons

One of the major advantages of our approach is that it provides additional duplications to the ones detectable by other traditional methods. It is able to bring to light smaller duplication fragments, otherwise hardly noticeable, which belong to a bigger, thus more important duplication block. By doing these, it ensures that the refactoring decisions are made with improved comprehension of the big picture, *i.e.*, it provides support for proper refactoring. Furthermore, the flexibility provided by means of the thresholds can lead to customized detection methodologies, that fit particular maintenance focuses.

As for the drawbacks, the tool is not capable of detecting renamed variables, due to its rather high granularity of comparison, although some of those could be found under modified duplication chains (with lower precison).

5.2 Future Work

To address the problem of course granularity, we think that a fuzzy comparison would make an improvement, giving the tool the advantages of a classical token-based duplication detector along with its novel duplication chain recovery approach. We would also be interested in researching on the information we could extract from the signatures of code duplication chains. Finally, while we think that it would be possible to associate some patterns in the signatures of duplication chains, it would be a real challenge to provide some assistance in the refactoring process, based on some identified patterns.

6 Acknowledgments

This work is supported by the Austrian Ministry BMBWK under Project No. GZ 45.527/1-VI/B/7a/02. We would also like to thank the LOOSE Research Group (LRG) for being such a great and challenging team.

References

[1] Brenda S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992.

[2] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings ICSM 1998*, 1998.

[3] Stefan Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, Universität Stuttgart, September 2002.

[4] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *J. Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[5] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, September 1999.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(6):654–670, 2002.

[8] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01)*, pages 301–309. IEEE Computer Society, October 2001.

[9] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, pages 244–253, 1996.

[10] F.Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *International Workshop on Principles of Software Evolution (IWPSE)*, pages 126–130, 2003.

[11] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proceedings Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pages 327–336, Gold Coast, Australia, December 2002. IEEE.

An Empirical Study on Software Defect Fix Effort Estimation with Incomplete Historical Data

Hui Zeng
Member, *IEEE*
School of Information Technology and Engineering, George Mason University
Fairfax, VA 22030, USA
hzeng@gmu.edu

David Rine
Senior Member, *IEEE*
Department of Computer Science, George Mason University
Fairfax, VA 22030, USA
drine@cs.gmu.edu

Abstract

Software defects fix effort is an important software development process metric in software development and corrective maintenance. Generally, parametric effort estimation techniques such as historical Lines of Code (LOC) and Function Points (FP) data are used to estimate effort of defects fix. However, these techniques require adequate complete historical data and fail in effectiveness when specifically estimating defects fix effort prior to starting new projects. This paper present an empirical study on specifically estimating software defect fix effort by a new non-parametric technique applying dissimilarity matrices and self-organizing neural network using software defects clustering and effort prediction with incomplete historical data.

KEY WORDS
Defects fix effort, software process metric, neural networks, Self Organizing Map (SOM), domain incomplete

1. Introduction

Software defects detection and removal are very important activities in software development and corrective maintenance. Earlier defect removal is more cost efficient than later [1]. Techniques for identifying when defects need to be removed at minimal software development effort while still providing customers with high quality are needed.

Recently research has shown special interest in estimation of defects fix effort [2],[3]. Most general techniques applied to estimate software development effort use parametric project size techniques incorporating Lines of Code (LOC) and Function Points (FP) that are based on specific kinds of historical data. However, these estimation techniques do not perform well when they are used to estimate defects fix time [4]. The main reason is that there are no predictable relationships between project size and defects fix effort/time. Numbers of defects in different domains require different defects fix effort/time. Moreover, parametric techniques require adequate historical data, and they fail to offer much help when estimating defects fix effort prior to beginning a different new software project without enough historical data.

Neural networks as one category of non-parametric techniques are usually suggested when estimating with incomplete historical data [5]. In this paper, we present a non-parametric estimation solution using self-organizing neural networks that can handle some binary, numerical, and nominal input data categorized in loosely-structured free text for defect fix effort estimation.

The background of software defect fix effort is introduced in Section 2. We present the detailed design of our experiment in Section 3, and the corresponding results are described in Section 4. Conclusions are provided in Section 5.

2. Background

Software defects fix effort is an important software development process metric that plays a critical role in software development and corrective maintenance. Estimation techniques with better accuracy of software defect fix effort estimation can help an organization to improve forecasting prior to software application development and maintenance, avoiding corrective maintenance-related project cost (effort) over-budget, under-budget and inaccurate financing.

Existing effort estimation techniques can be categorized as parametric and non-parametric. Parametric techniques hypothesize the structure of a model for a system of interest, with several parameters left to be tuned by empirical data [12]. The advantage for a parametric model is that it is sometimes easier to use; some statistical techniques can be applied along with the model, and techniques do not required any initial training for the system before estimation. The drawback is that it needs strong understanding of historical data. Moreover, some tuned parameters are based on certain software development environments and cannot be easily transplanted to another new project. Nonparametric (distribution free) techniques are not solely based on statistics, but on discrete objects in a feature domain.

These are techniques to build models that rely heavily on the use of data without using many prior suppositions [5]. Neural Networks (NNs) belong to nonparametric techniques. The advantages of a non-parametric model are initial training before actual estimating begins, less understanding required for input data, a self-adaptive system, and improvements when using incomplete data. The drawback is that NN is sometimes not easy to represent and fewer statistical techniques are applied.

Generally parametric estimation techniques using historical LOC [3],[4],[11] and FP [10] data are traditionally applied to estimate software development effort. However, these techniques do not perform well [5] because defects fix effort is based on counts of defects in each development phase and different domains. It is not easy to compare counts of lines of rewritten code or FPs for a new project with fixing defects effort when code will be written within the context of a different organization. Another reason is the relationship between defect fix effort and LOC and FP are not as simply as people expect. Different bugs from different development phases and different scenarios may cause different fix efforts.

Usually, the collected historical data in the real world could have certain features missed or misclassified, such as NASA IV&V Facility Metrics Data Program (MDP) data repository [7] dataset. The MDP static defects data contains defects data that remains constant throughout the life cycle of that defect. The challenge of a MDP data set is that defects fix effort is only based on each actual defect, not based on each type of defect. Moreover, there are no rigorous categories for these defects; they are only categorized in loosely-structured free text (not clearly categorized or defined in a proper domain) [6]. Moreover, these collected data is a mixture of binary, numerical, nominal, interval, and ordinal data in the real world. This kind of mixture of historical data has seldom been considered in existing effort estimation techniques.

In this paper, we present a new framework for non-parametric defect fix effort estimation that can deal with mixture of various input data categorized in loosely-structured free text data. Binding symbolic data manipulation with a Self-Organizing Map (SOM) Neural Networks (NN), our proposed framework provides better support for software defect fix effort estimation with incomplete historical data, as well as a post-processing probabilistic model that can compensate for the weakness of NNs in the area of statistical analysis.

3. Experiment Design

Our methodology for software defects fix effort framework is depicted in Fig. 1. The system architecture consists of three components: (i) feature extraction, (ii) self-organizing maps, and (iii) probabilistic measurement. In our experiment, we select the metrics summarized in Table 1 as our estimation input variables.

Fig. 1. Framework Methodology Architecture

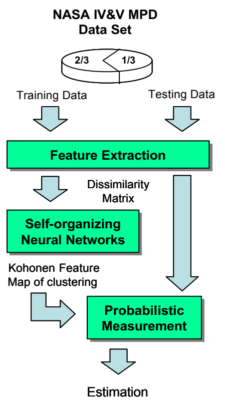


Table1. Input Variables for Defect Fix Effort Estimation

Variable	Description	Types
1..`Fix_Hour`	The actual number of man hours the fix took to implement	Integer
2.`Severity`	The severity of the defect	1,2,3,4,5
3. `How_Found`	The stage in which the defect was found	Acceptance Test, Analysis, Customer Use, Engineering Test Inspection, Mission Critical, Planned Test, Regression Test, Release_I&T
4. `Mode`	The mode the system was operating in	DEV02, DEV03, DEV04, OPS, Other, TS1,TS2
5. `Problem_Type`	Specific reason for closure of error report	Configuration, COTS/OS, Design, not a bug, source code
6. `SLOC_COUNT`	The actual number of SLOC changed or added	Integer

3.1 Feature Extraction

As introduced earlier, some data metric measures related to defects fix effort are symbolic data categorized in loosely-structured free text, which cannot be used by NN directly as input vectors, and need to be converted to binary data. We first converted these nominal features to binary features [13], second applied the Jaccard coefficients method, and then generated a dissimilarity matrix so the neural network can accept as one of input vectors. In this experiment, four nominal variables: `Severity`, `How_Found`, `Mode`, and `Problem_type`, were converted to binary variables [13]. A contingency table shown in Fig.2 for binary data type was derived.

Fig. 2. The Contingency Table for Binary Variables

		Sample *j*		
		1	0	*sum*
	1	*a*	*b*	*a+b*
Sample *i*	0	*c*	*d*	*c+d*
	sum	*a+c*	*b+d*	*p*

An asymmetric dissimilarity d_k was then carried out using Jaccard coefficients.

$$d_k(i,j) = \frac{b+c}{a+b+c}$$

where $d_k(i, j)$ is the dissimilarity score between ith and jth samples with respect to the k^{th} input variable [13].

In this experiment, the sixth variable, ***SLOC_COUNT***, is an interval-valued variable whose value was normalized between 0 and 1. Manhattan distance was used to compute another dissimilarity matrix. For m samples, $m(m\text{-}1)/2$ dissimilarity vector matrices with two attributes are generated.

NASA MDP datasets KC1 and KC3 were used in our experiment as a test bed to assess the performance of the estimation methodology. KC1 is the development with a single computer system component item (CSCI) within a large ground system, which is made up of 43 KSLOC of C++ code [7]. KC3 is an effort that has been coded in 18 KLOC of Java. The purpose of the code is the collection, processing and delivery of satellite metadata [7].

A total of 947 samples corresponding to 36 different software defects fix efforts were used in our experiment, which were randomly divided into two data sets - 631 samples for training the self-organizing map (SOM) neural network and the remaining 316 samples for validating the estimation performance of the SOM. Two attributes of dissimilarity measurement derived from normalized *SLOC_COUNT* and Jaccard coefficients using four nominal variables are the SOM inputs.

3.2 Self Organizing Maps (Kohonen Networks)

Kohonen SOM [8][9] is a sheet-like artificial NN, the cells of which become specifically tuned to various input variable patterns or classes of patterns through an unsupervised learning process. In other words, it can be said that the SOM algorithm is an unsupervised learning algorithm that creates topological mappings between the input data and map units: If two input patterns are similar, then the most active units responding to the two input patterns are located near each other on the SOM. The locations of the responses tend to become ordered as if some meaningful coordinate system for different input features were being created over the NN. The spatial location or coordinates of a neuron in the NN match up to a particular domain of input patterns. It is this feature that is of particular interest since we need to figure out some pattern with different samples. Therefore the defects fix effort in terms of the variable, *Fix_Hour*, in the unseen data, can be estimated. After the network training, all training samples were clustered and the probability distributions of *Fix_Hour* within clusters were derived.

3.3 Probabilistic Measurement

In order to evaluate the performance of our estimation effort prediction model, we used magnitude of relative error (MRE) [5] as our evaluative measure:

$$MRE_i = \frac{|\,ActualEffort_i - \mathbf{Pr}\,edcitEffor_i\,|}{ActualEffort_i}$$

3.4 Experiment Setup

As introduced in the previous section, the collected historical data come from two different types software development environments: KC1 produced a defects dataset with C++ programming developments, while KC3 is based on Java programming developments. Two hypotheses were provided and tested to support our estimation experiment:

H1: The framework provides improved estimate performance if training and testing historical data are from similar environment (programming language, domain applications, and development organizations are similar).

H2: The framework provides improved estimate performance if training and testing historical data are from different software development environments.

Two experiments described in Table 2 are setup to validate the above two hypotheses.

Table2. Experimental Design

	Dataset Name	Training Datasets	Testing Datasets	Cross Validation
Part 1	MDP KC1	631 from KC1	316 from KC1	Yes
Part 2	MDP KC1 and KC3	631 from KC1	68 from KC3	No

4. Results

As introduced, the training data group was used to train the weights of self-organizing NNs. To cluster the training samples, self-organizing NN with 25 hidden neurons in a hexagonal 5-by-5 network topology was initialized. Based on our observation, a 5x5 topology were sufficient for covering the feature space.

Self-organizing map (SOM) consists of a single layer with the weight from the input mapping to 25 neurons. The training process repeated to minimize the networks' quantization error by adjusting weights till the error was significantly small. The weight vectors were originally located in the midpoint of feature map by assigning identical initial values. During the training phase, the neurons have started to move toward the various training groups and the feature maps simultaneously update the weights of the winner node (neuron) and its neighbours. The result is that neighbouring neurons tend to have similar weight vectors and to be responsive to similar input vectors. The weights of feature map were learning to categorize their input, also learned both the topology and distribution of their input. After the network was well trained, these samples were clustered into neurons to form a feature map.

After SOM training, the known values of defects fix effort represented by variable ***Fix_Hour*** were assigned to the found clusters. The probability distribution corresponding to various ***Fix_Hour*** values within each cluster was derived. 61 testing samples from KC1 and 613 defects data from KC3 were then used to validate overall performance. Testing samples followed the same procedure as training samples, such as going through feature extraction and finding a set of dissimilarity vector. During the testing phase, each unseen sample was compared to all training sample vectors to generate 947 dissimilarity vectors. These vectors were fed into already trained self-organizing neural network and produced an unknown probability distribution. We then compared this unknown distribution against known probability distribution of each cluster. For the cluster that offers the most likelihood, it is selected as the best response to the `Fix_Hour` inquiry.

We then use MRE as the evaluative measure to evaluate the performance of our estimation effort prediction model. As the histograms of defects fix effort can be grouped as 6 groups as shown in following table, we calculated average MRE and maximum MRE within each histogram. Table 3. shows the performance evaluation using data samples from KC1.

Table 3. Performance Evaluation Using KC1 Dataset

Measurement of Accuracy	Range of Defect Fix Effort (Man-hours)					
	0-1	2-8	9-20	21-50	51-80	81-200
Average MRE	6%	14%	37%	51%	41%	37%
Maximum MRE	93%	331%	251%	165%	108%	90%

We also evaluated the estimation performance by using another NASA MDP dataset KC3 as our other testing data. KC3 is a metrics dataset with projects of Java developments. 68 defects data samples of KC3 are used for validation. The results are shown in Table 4.

Table 4. Performance Evaluation Using KC3 Dataset

Measurement of Accuracy	Range of Defect Fix Effort (Man-hours)					
	0-1	2-8	9-20	21-50	51-80	81-200
Average MRE	40%	130%	181%	182%	95%	159%
Maximum MRE	881%	520%	213%	181%	216%	186%

From the above result, we found the average MRE is from 6% to 51% and the maximum MRE is from 90% to 331% by using dataset KC1, which indicates that the performance of estimation by using our method is robust except for the bin between 21 to 50 man hours. Performances within other ranges are less than the excellent effort estimations norm of 42%. So Hypothesis 1 is accepted. However, when we evaluate the estimation performance by using KC3 68 defects data as testing data, a poorer estimation result is derived, the average MRE is from 40% to 182%, and the maximum MRE increases from 181% to 881%. Therefore Hypothesis 2 is rejected.

The main reason of different estimation performances is because the KC3 produced a defects dataset with Java programming developments of one application type involving data processing in satellite metadata, and KC1 produced a defects dataset with C++ programming developments of a different application type that includes a single CSCI within a large ground system. As two different development environments can cause different software requirements, analysis, designs, coding, and testing strategies, corresponding defects fix effort may differ. So, even if we applied non-parametric estimation techniques like NNs with the feature of self-adaptation, if the environments of software development are totally different, we still cannot always anticipate better performance. Estimation techniques only perform consistently well in the family oriented software development environment like product line software development.

5. Conclusions

In this paper, we present our strategic solution to estimating software defects fix specific effort by using dissimilarity matrix and self-organizing NNs for software defects clustering and effort prediction instead of traditional development wide project size techniques such as Lines of Code (LOC) and Function Points (FP). By using our new estimation method, defects fix effort/time can be estimated using the number of defects in various domains instead of fixed domains, using functions with self-adaptive features. The experimental results indicate improved performance when applied to estimates for similar software development projects.

References:

[1] B. W. Boehm, Software Engineering Economics: Prentice Hall, 1981.

[2] A. Mockus, Understanding and Predicting Effort in Software Projects, *25th International Conference on Software Engineering*. May 03 - 10, 2003

[3] B. Boehm, et al. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, November 1995.

[4] K. Manzoor, A Practical Approach to Estim Defect-fix Time, http://homepages.com.pk/kashman/

[5] M. R. Lyu, *Handbook of Software Reliability Engineering*(McGraw Hill, 1996).

[6] T. Menzies and R. Lutz, Better Analysis of Defect Data at NASA, *The 15th Intn'l Conf. on SE and Knowledge Engineering*, July, 2003.

[7] NASA MDP site http://mdp.ivv.nasa.gov/

[8] T. Kohonen, The Self-Organizing Maps, *Proceedings of the IEEE*, 1990 78, 1464-1480

[9] T. Kohonen, Self-Organizing Maps, Springer Series in Information Sciences, New York, 1997

[10] N. Fenton, "Software Metrics", Second Edition, ITP, 1997

[11] S. Chulani, Bayesian Analysis of Software Cost and Quality Models, Ph.D Dissertation, Univ. of South California, 1999

[12] R. Kennedy, Y. Lee, *Solving Data Mining Problems through Pattern Recognition* (Prentice Hall, 1997)

[13] D. Hand, H. Mannila, and P. Smyth, Principles of Data Mining, The MIT Press, 2001.

Experience Report on Maintaining Executable UML Software

Ned Chapin
InfoSci Inc., Box 7117
Menlo Park CA 94026-7117
NedChapin@acm.org

Abstract

Raising the level of abstraction at which the personnel work is a long-established approach to improving software personnel productivity, in both software development and software maintenance. The object-oriented (OO) technologies are one example of technologies promoted to raise the level of abstraction in software analysis and design. An outgrowth of object-orientation, executable UML, has been advocated as a way of bringing a lift in the level of abstraction into the implementation and maintenance phases of the software life cycle. Based upon an industrial case that fits MDA (model-driven architecture) well, this paper provides an experience report on software main-tainability effects from using executable UML. This paper concludes with a listing of five lessons learned, and the observation that the reported experience with executable UML showed no improvement in main-tenance over ordinary OO experience at the site studied.

1. Site studied

Continuous evolution (the ICSM 2005 theme) characterizes the job shop manufacturing plant studied and reported on here, because the plant does assembly work mostly on a short-term contract basis for a mix of customers from various industries. The jobs are mostly the manufacture of assemblies from components and subassemblies made elsewhere, and the resulting assemblies are in turn shipped elsewhere. The plant's management has elected not to use ERP (enterprise resource planning) packaged software for its job shop production floor operations. Instead, management elected to develop in house and maintain in house custom software using object-oriented (OO) technology including using the UML (unified modelling language). The plant's shop floor has multiple production lines with work stations that are reconfigured frequently as customers and their needs change.

The character of the assembly processes makes MDA (model-driven architecture [1]) attractive to the plant's management for the design of the software systems. The software systems ("Line Systems") used to operate the lines are all different since each line has unique characteristics, both as to many of the operations at some work stations, and as to material ("cargo") movement between work stations. The Line System software used with line "L6" is the plant's pilot implementation of executable UML [2], now in its second year. One instance out of the seemingly continuous flow of changes to that L6 Line System is the archetypical enhancive maintenance task highlighted here [3].

That maintenance task initiated by the plant's Safety Officer dealt with cargo movement. This enhancive task was approved by the plant management and the information technology (IT) management, and funded for two stages. The first or test stage (reported on here) was to involve only one turntable of the eight four-position turntables of the 25-segment L6 line. For testing, turntable D on the L6 line was chosen. The task was for the addition of two safety stops ("catches" A and B) per turntable (Figure 1).

The three justifications for the Safety Officer's request were the possibility of undetected cargo on the line, the possibility of non-existent ("ghost detection") cargo, and the possibility of inconsistent (contentious) movement of cargo between segments. The L6 production line with its sixteen stub segments, routed work in progress to and from the sixteen work stations and two end stations ("Begin" and "End"), and provided temporary storage between work stages for some work in progress. Cargo motion along any part of the line could be in either direction, and different segments could be independently and concurrently moving cargo in any direction, or be idle (stopped).

In executable UML, the production line situation provides a congenial context for having the model drive the software design, as advocated in MDA [1], but using executable UML was the emphasis with MDA as a handmaiden, not the reverse. For example, by the requirements specified by the Safety Officer, under what conditions should a turntable's catches be up (raised position) and under what conditions should they be down (lowered position)? What are the error situations—i.e., the "impossible" circumstances that (should) never exist—and how are they detected and the situations handled?

What about the timing of behaviors—when should and should not they happen? Again, the production line situation provides a context congenial for executable UML because the people who work with the line on a daily basis, think, talk, and act based upon their assessment of the states of portions (segments) of the line. Thus, on a particular line segment, the state is particular cargo in a forward motion, or on another segment, the state is idle (stopped) with cargo not moving as when the line segment for the next cargo movement direction is currently handling other cargo. Some of the states have association relationships, as exampled in Figure 2. Executable UML relies heavily on the use of statecharts, and requires the recognition in the model of the existence of states, and of their initiation, termination and sequence.

Executable UML has supporting software that (typically) generates C or C++ code as an intermediate to be compiled to produce object code [2]. While the precise action semantics (PAS) of executable UML can be inserted partly into the class rectangles in UML class diagrams, mostly is inserted into the state rectangles in UML statecharts [2]. Figure 3 is a preliminary statechart including the precise action semantics (PAS) of executable UML for the new `Catches` object of the enhancement to the L6 Line System. Ignored in executable UML are aggregation and composition indications [2]. Typically using executable UML also results in refinements and supporting content additions in the associated (if any) UML sequence diagrams, activity diagrams, use case diagrams, and collaboration diagrams to document the maintenance.

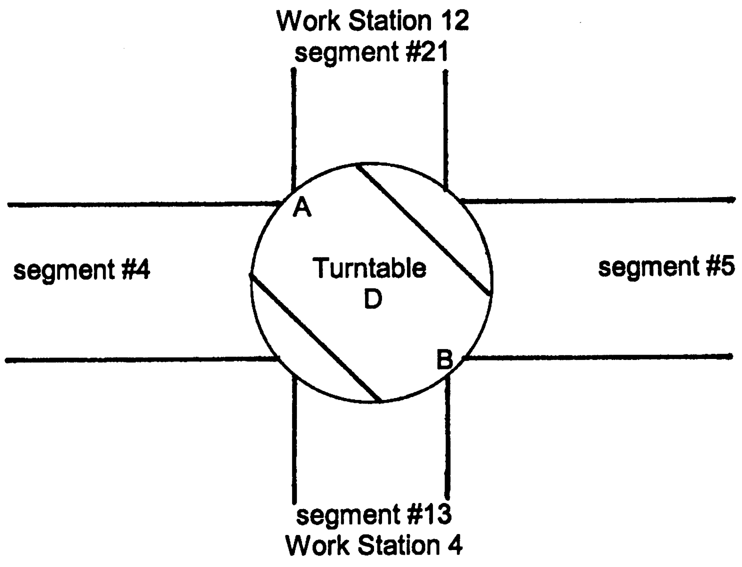


Figure 1. Diagram of turntable D and its position with respect to the associated line segments and work stations. A and B mark the new safety catches.

2. Management concerns

Because the plant makes changes often daily to the systems (such as the Line Systems) it uses for running the plant's operations, maintainability is of special concern to the plant's production management. Therefore, the maintenance experience from using executable UML has been of particular interest to the plant's information technology (IT) management—should the IT management kill its use or extend its use? Maintenance of executable UML software can be done by studying and revising the executable UML,

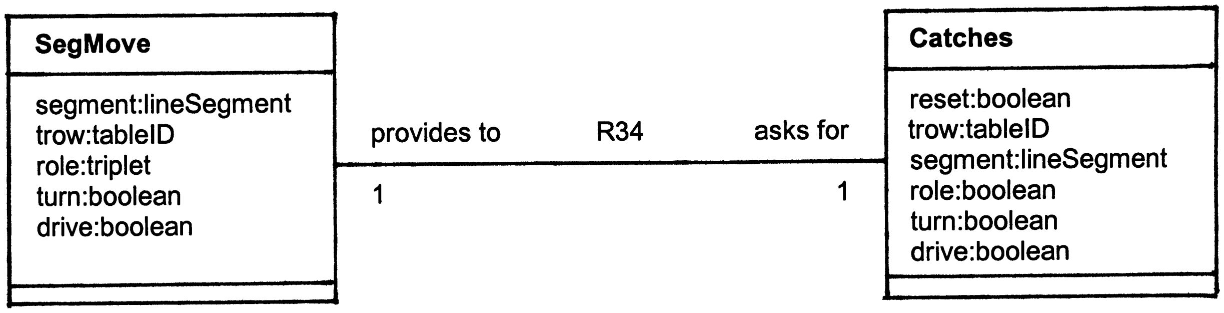
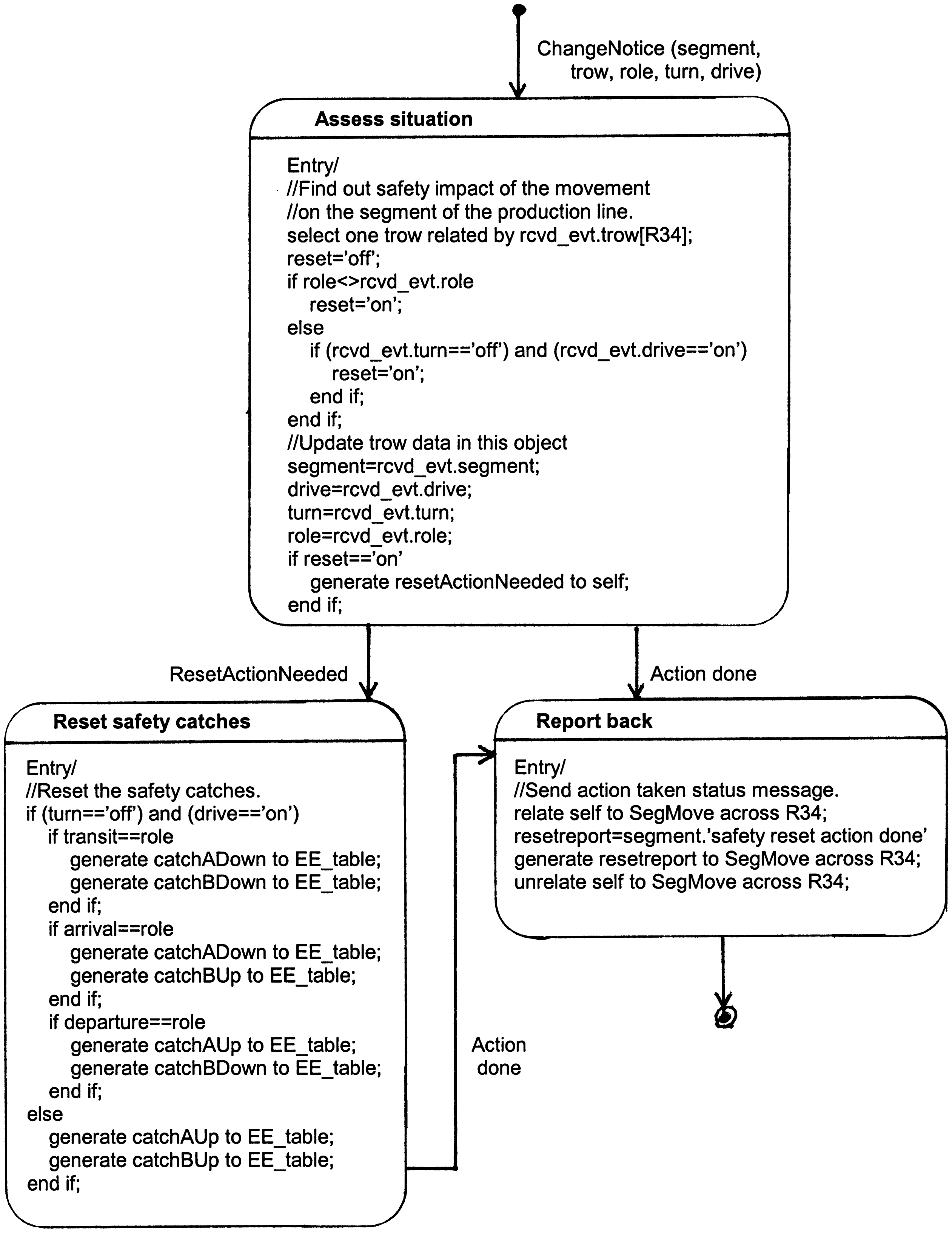


Figure 2. Detail on two objects in the L6 Line System class diagram, listing the attributes for the added object Catches, the new attributes for the object SegMove, and the new relationship (R34).



```
Entry/
//Find out safety impact of the movement
//on the segment of the production line.
select one trow related by rcvd_evt.trow[R34];
reset='off';
if role<>rcvd_evt.role
   reset='on';
else
   if (rcvd_evt.turn=='off') and (rcvd_evt.drive=='on')
      reset='on';
   end if;
end if;
//Update trow data in this object
segment=rcvd_evt.segment;
drive=rcvd_evt.drive;
turn=rcvd_evt.turn;
role=rcvd_evt.role;
if reset=='on'
   generate resetActionNeeded to self;
end if;
```

```
Entry/
//Reset the safety catches.
if (turn=='off') and (drive=='on')
   if transit==role
      generate catchADown to EE_table;
      generate catchBDown to EE_table;
   end if;
   if arrival==role
      generate catchADown to EE_table;
      generate catchBUp to EE_table;
   end if;
   if departure==role
      generate catchAUp to EE_table;
      generate catchBDown to EE_table;
   end if;
else
   generate catchAUp to EE_table;
   generate catchBUp to EE_table;
end if;
```

```
Entry/
//Send action taken status message.
relate self to SegMove across R34;
resetreport=segment.'safety reset action done'
generate resetreport to SegMove across R34;
unrelate self to SegMove across R34;
```

Figure 3. Preliminary statechart for the Catches object.

and then regenerating the intermediate code before recompiling. The maintenance work rarely requires studying and revising the intermediate (usually C or C++) code. The field experience elsewhere thus far in using executable UML has been insufficient to provide a quantitative basis for confidently determining executable UML's effects on personnel productivity even in software development, let alone in software maintenance. Also, the precise action semantics are downplayed in UML 2.0 [4]. The effects of executable UML on the "...ilities" of software are certainly relevant because of the "...ilities" connection to maintainability [5], but thus far published reports of studies are in very short supply. Hence, the plant's management is interested in what are the lessons learned thus far.

3. Lessons learned

The question of interest to the plant's management is to how does the more than a year of maintenance experience on the L6 line with executable UML compare with the maintenance experience on the other lines not using executable UML? Informal surveys of the staff involved, a review of available documentation, and an evaluation of the source code of selected projects, point to five lessons learned and one general conclusion. The lessons learned are these:

- *Dependencies must be made explicit*. In executable UML, the objects are dormant until activated, run concurrently with other objects but independently, run to their own completions, and then go dormant to await their next activations. While running, objects may send messages, and may receive both backlogged and current messages. What the objects are sensitive to, and what interactions they have and may have with other objects are specified in executable UML within the objects. Hence, maintenance in executable UML has to give explicit attention to dependencies among objects.
- *Use of association classes and of subclasses is encouraged*. Adding subclasses and association classes is convenient and expeditious when using executable UML in maintenance. Association classes and subclasses are not unique to executable UML, and their use is generally regarded well in theory, but not as well in practice in using OO technology.
- *All attributes must be single-valued*. In the maintenance of systems using OO technology, multiple-valued attributes are sometimes inserted, but that practice is not commonly recommended. With executable UML, single-valued attributes are the required way to go in maintenance.
- *Dynamic creation and deletion of objects is discouraged*. While they are as permitted and as useful in executable UML as they are in OO software generally, they turn out to be potential bug sources during maintenance in executable UML. The reason is that a dynamically created or deleted object has dependencies that have to be made explicit. While their creation is cumbersome in maintenance with executable UML, their deletion is worse in practice, because any undeleted dependencies can and may give rise to hard to locate errors.
- *Documenting ripple becomes more important*. When maintaining software in executable UML, changes usually must also be made in other parts of the software than the part being concentrated upon, because dependencies need to be explicit. The temptation is to skimp on updating the UML documentation of affected parts of the software. If an organization's procedures and software tools used tolerate this, then the UML description of the system becomes increasingly obsolete more rapidly. This can make future maintenance slower and more costly to do, especially when the usual OO technology is used along with executable UML.

The general conclusion is that overall, the maintenance experience with executable UML and the associated MDA at this plant has thus far been indistinguishable from that for similar software using OO technology but not using executable UML. In qualitative terms, maintenance tasks appear to be estimated for and use about the same calendar time and person time, when executable UML is used, as when it is not used in doing the tasks. Thus far, the plant management has not extended beyond the L6 line its use of executable UML.

4. References

[1] OMG, *MDA Guide Version 1.0.1*, Object Management Group, Inc., Needham MA, 2003.

[2] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundations for Model-Driven Architecture*, Addison-Wesley: Boston MA, 2002.

[3] N. Chapin, J. E. Hale, K. Md. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance", *Journal of Software Maintenance and Evolution: Research and Practice,* John Wiley & Sons, Ltd., Chichester, UK, January-February 2001, pp. 3–30.

[4] K. Scott, *Fast Track UML 2.0*, Springer-Verlag, New York NY, 2004.

[5] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability", *Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, 1992, pp. 337–344.