2009 Sixth International Conference on Information Technology: New Generations

An Eclipse Plugin for the Automated Reverse-Engineering of Software Programs

Philippe Dugerdil, David Kony, Javier Belmonte
Department of Information Systems
HEG-Univ. of Applied Sciences,
7, route de Drize, CH-1227 Geneva, Switzerland
{philippe.dugerdil, david.kony, javier.belmonte}@hesge.ch

Abstract

In the reverse engineering of a software program, one of the key difficulties is actually to understand the software. While the published techniques work top down or bottom up, our approach works middle-out: before trying to understand the low level code, we first rebuild a hypothetical analysis model from the use-cases of the system. This model then represents the target of the understanding task. In fact we try to map the code elements to the analysis objects. For this approach to be useable in large industrial software systems, it must be supported by a powerful tool. This paper presents the Eclipse plugin we developed to support our methodology, as well as a reverse engineering scenario using this tool. We then discuss the technology we used and the result we obtained.

Keywords – Reverse engineering, analysis tool, software understanding, dynamic analysis, Eclipse.

1. Introduction

To extend the life of a legacy system, to manage its complexity and decrease its maintenance cost, one option is to reengineer it. Recently, we developed a reverse-engineering process based on the Unified Process which rests on the dynamic analysis of program execution. The theoretical framework of our technique has been presented elsewhere [7][8]. The first experiments with this reverse engineering process have been performed by hand. Although these were encouraging, the size of real world industrial software asks for the support of a powerful tool. The goal of this paper is to present the tool we have developed as well as the way it can automate the most difficult task of the process: the mapping from low the level source code elements to the analysis model elements. In the following text, section 2 presents a short summary of our methodology and section 3 justifies our approach with respect to the software understanding effort. Section 4 presents the engine that maps the source code elements to the analysis model elements and section 5 present the tool itself with its user interface. Section 6 presents a reverse engineering scenario that uses the tool and section 7 discusses the results obtained so far and the future work. Section 8 presents the related work.

2. Summary of our methodology

Generally, legacy systems documentation is at best obsolete and at worse non-existent. Often, its developers are not available anymore to provide information of these systems. In such situations the only people who still have a good perspective on the system are its users. In fact they are usually well aware of the business context and business relevance of the programs. Therefore, our iterative and incremental methodology, which is based on the Unified Process [11], starts from the recovery of the system use-cases from its actual users. Its main steps are [8]:

- Re-documentation of the system use-cases;
- Design of the analysis models associated to all the use-cases;
- Re-documentation of the visible structure of the code;
- Execution of the system according to the use-cases and recording of the execution trace;
- Analysis of the execution trace and identification of the classes involved in the trace;
- Mapping of the classes in the execution trace to the objects of the analysis model.
- Re-documentation of the architecture of the system by clustering the classes based on their role in the use-case implementation.

978-0-7695-3596-8/09 $25.00 © 2009 IEEE
DOI 10.1109/ITNG.2009.73

In the absence of any documentation on the system to reengineer, the Unified Process' analysis model associated to each use-case represents our best hypothesis on the actual architecture of the system. Figure 1 presents an example of an analysis model with the stereotypical classes (analysis object) that represent software roles for the classes. These roles are: the boundaries (interface with the outside world, i.e. screens), the entities (information containers) and the control objects (coordinators of the use-case execution) [11].

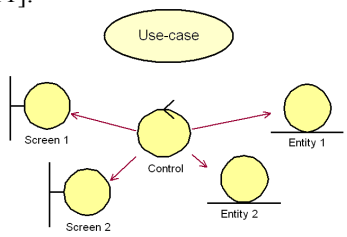

Figure 1. Use-case and analysis model

Besides, we must re-document the *visible* structure of the code based on syntactic clues such as the modules, packages and classes declarations, as well as the directory structure in which the elements of the code are stored. This let us identify the code element that we must understand. Therefore, as the next step, we must find the classes in the actual implementation that play the roles of the objects in the analysis model. Then, we run the system according to each use-case and record the execution trace i.e. the functions and procedures called during execution (Figure 2).

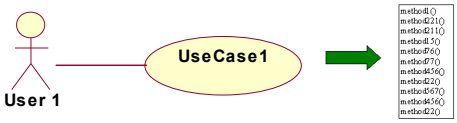

Figure 2. Use-case and the associated execution trace

Next the functions and procedures called, recorded in the trace, are linked to the classes or modules they belong to. These represent the classes and modules that actually implement the use-case. Then the source code of these functions and procedures is analyzed to find evidence of database access and screen display. The classes and modules containing database access functions will be mapped to entities and the ones containing screen display functions to boundaries [8]. The remaining classes are mapped to control classes. At this step, we know the role of the classes in the implementation, but not the exact analysis objects they can be mapped to. To perform this last step, we analyze the sequence of involvement of the analysis objects in the use-case and compare it to the sequence of occurrences of the identified implementation classes in the execution trace. In fact, when analyzing a use-case, one must identify at each flow step the analysis object involved [11]. Consequently, the sequence of action steps in a use-case flow leads to a sequence of analysis object involvements (Figure 3).

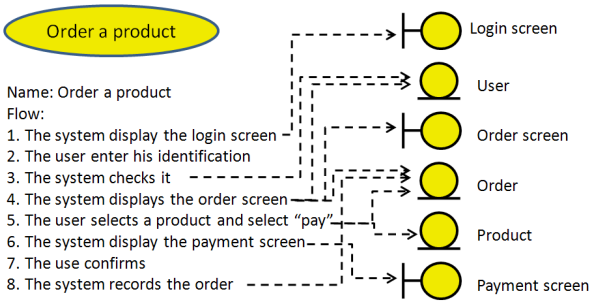

Figure 3. Use-case and involved objects

After having compared the sequence of analysis object involvements to the sequence of implementation classes occurrences, we can map the objects as showed in Figure 4. Moreover, to help with the mapping, we also compare the associations in the analysis model to the ones between the implementation classes. The last step in our method is to recreate the high-level architecture of the software by clustering the implementation classes according to the use-case they implement and to the role they play.

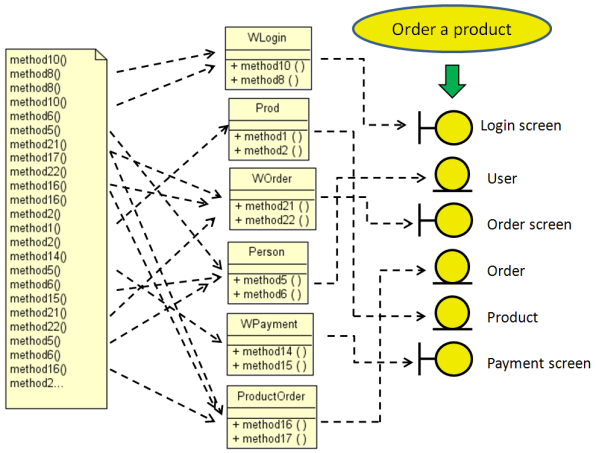

Figure 4. Mapping implementation classes to analysis object based on sequence

3. Software understanding justification

Software understanding theories have long been reported in the literature [1][2][13][14][17][18]. Generally the authors distinguish between top down (from the knowledge of the functional requirement down to the code) and bottom up (from the code to the

function it implements). However, few theories have been proposed for model-based program understanding that we could classify as middle-out. In our approach, the maintenance engineer would first rebuild the analysis model of the use-cases before trying to understand the code. This model represents the target of the understanding of the code, since the link between the functional requirements (use-cases) and the analysis model is straightforward. By so doing, we move the a-priori functional understanding of the system closer to the code (i.e. we "transfer" the functional understanding from the use-case model to the analysis model that is closer to the code). Therefore, the gap to fill to understand the code is smaller. This is exactly what our integrated environment is able to do. In fact, the mapping of the implementation classes to the analysis objects creates a link between the use-cases and the implementation classes. However, it is important to note that a single implementation class can be involved in the implementation of several analysis objects. Moreover, a single analysis object can be implemented by several classes. In short, the mapping between analysis object and implementation classes is many to many. Often, we can associate some of the methods of the implementation classes to each analysis object they implement. It is therefore important to know which methods in the execution trace the mapping from one analysis object to an implementation class rests on. Finally it is also required for the environment to let us freely navigate between all the models and information source and to be able to highlight the corresponding elements in all the models and information sources.

4. Automating the mapping

In fact, for any reasonable size industrial system, the mapping between the analysis objects and the implementation classes cannot be done by hand because of the number of classes involved and the size of the execution trace. Therefore, to automate this mapping, we designed a production system where the production rules implement the heuristics we developed when applying our methodology by hand [1]. However, since the mappings inferred by the heuristics are probable but not certain, we had to complement the production system with a Truth Maintenance System TMS [6] to deal with the incertitude of the inferred facts. In short, a TMS can be seen as a graph whose nodes are the inferred facts and whose edges are the inference dependencies between the facts. When the certainty value of a given fact is modified, this value is propagated to all the dependent facts in the graph to maintain the global coherence of the inferred facts.

Since the production rules must process the use-case flow, the analysis model, the source code and the execution trace, we need an integrated environment where all these models are available and linked. This is summarized in Figure 5. Because the mappings rest on the analysis of the execution trace i.e. on the sequence of method calls, we can trace for each successful mapping the methods that lead to it. In summary, our tool will record the links between:

- the use-case;
- the execution trace that is generated when executing one scenario from the use case;
- the analysis model corresponding to the use-case;
- the implementation classes that correspond to the object of the analysis model;
- the methods in the implementation classes that lead to the mapping.

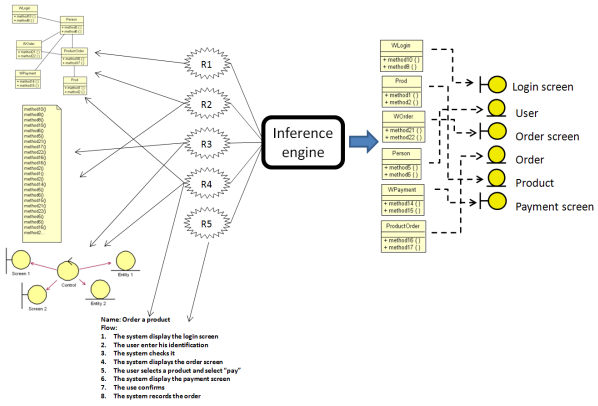

Figure 5. Inference engine to infer the mapping

To support our methodology, we need an integrated tool that is able to display all the models and information sources as well as record and highlight the links between all the corresponding elements. Interestingly enough, one of the most advanced software engineering tools on the market, RSA (Rational Software Architect) available from the leader in the implementation of the Unified Process: IBM® is not able to represent the traceability links between these models and information sources. For example the objects of the analysis model cannot be *formally* linked to the corresponding design model classes and the classes of the latter cannot be *formally* linked to the corresponding implementation classes. By formally we mean that no mechanism maintains the bidirectional traceability constraints between these model elements. In fact, RSA adheres to the MDA (Model Driven Architecture®) approach from the OMG®. Then, it is able to generate a given model from another model

(normally the PSM from the PIM) by executing some transformation rule. But the generated models weakly refer to each other: we may know that model 1 has been generated from model 2 but the connection between the elements of each model is not recorded. However, this is exactly the kind of traceability we need to "understand" the software.

5. Reverse analysis Eclipse plugin

Our tool, which is developed as a plugin to the Eclipse platform, has five main components:

1. The extended file explorer;
2. The extended file editor
3. The analysis model editor;
4. The use-case editor;
5. The model mapper that includes the production system and the TMS.

Figure 6 presents the integrated reverse engineering environment in the Eclipse framework. Of the above 5 components, only 4 are visible in the picture: the model mapper runs in the background and does not have a specific view. On the top right, one sees the

the most suitable to be Violet [19]. The use-case editor is represented on the bottom. It has three subviews. On the left one represents the analysis objects involved in the use-case. These are all the objects present in the analysis model on the top right. In the center we find the use-case flow editor. On the right we show the analysis objects associated to the selected action step in the use-case flow. The objects represented on the bottom right are the one associated to the 6th action step in the use-case flow. The column "stat" gives the number of analysis objects associated to each action step. To our knowledge, this is the only tool that leverages the UP analysis discipline by linking the analysis objects to the action steps of the use-case.

6. Reverse engineering scenario

After having recovered the use-cases of the system, we redocument its visible architecture. In the case of Java programs, this can be done automatically through the use of a software engineering environment such as RSA. Then we instrument the source code of the system to be able to generate the execution trace. The

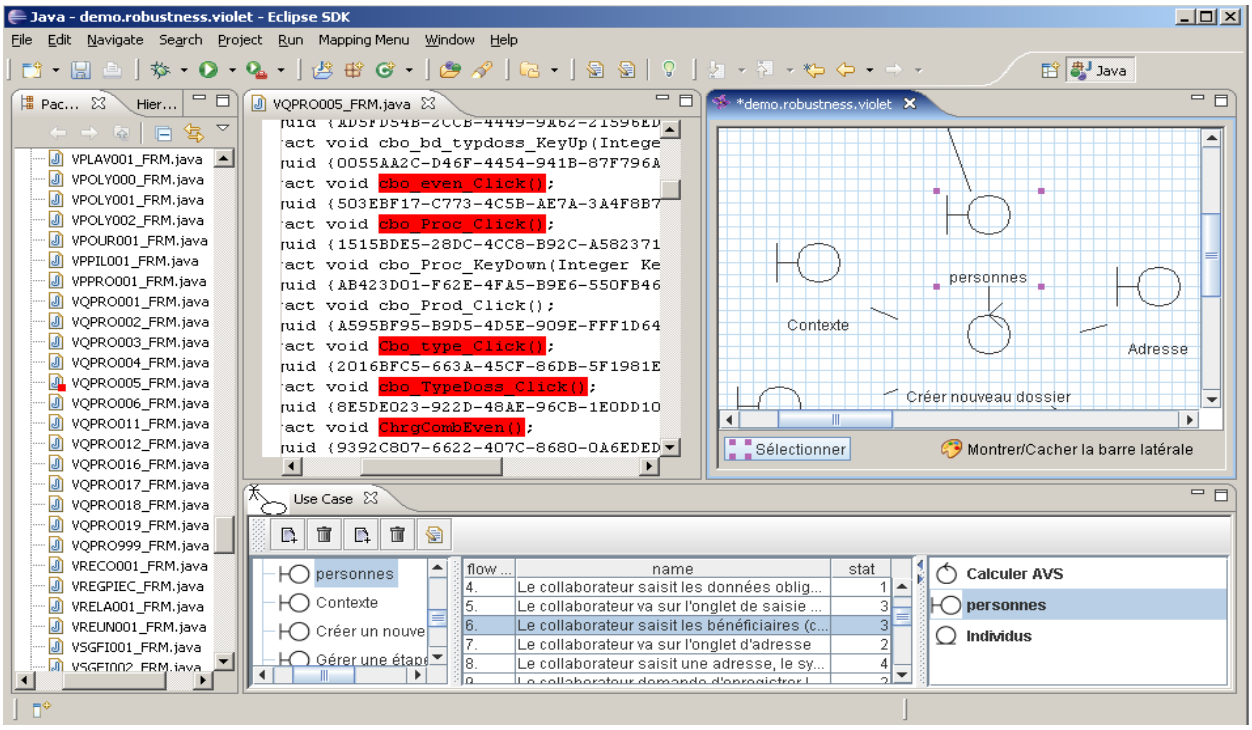

Figure 6. The reverse engineering environment under Eclipse

analysis model editor. This tool is an open source plugin that has been extended to include the analysis model stereotypes and the ability to broadcast the selected objects to the other views. We tried several open source UML diagrams editor tool and we found

instrumented code is compiled and run according to scenarios corresponding to the use-cases. The generated execution trace is then recorded. Once this preliminary work is completed, we can start to analyze the system with our tool. First we select the source

files of the system to analyze, through the file menu of the tool. Then, for each of the use-cases, we proceed as follows:

1. We enter the flow of events of the use-case by using the use-case editor of the tool.
2. We manually analyze the use-case and design its analysis model in the analysis model editor.
3. We attach each of the analysis objects of the model to the corresponding action step of the use-case flow. This is done by picking one or more of the available objects listed on the left in the use-case editor.
4. When this is done we can launch the object mapper. The latter then asks for the associated execution trace file to be used as input.
5. After the mapping is completed, the result can be displayed as annotations in the models and editors.

After having executed the mapper, if one selects one analysis object in the list on the left of the use-case editor (see figure 6) then:

1. The file explorer displays a little red dot to the bottom right of some of the file icon. These are the files containing the classes that are mapped to the selected analysis object.
2. When we open one of these files, the editor highlights the signatures of all the method that are involved in the mapping.

These represent the implementation classes that play the role corresponding to the selected analysis object. Similarly, the analysis object can be selected in the analysis model editor (top right), the resulting display will be the same. For example, in Figure 6, we selected the boundary "Personnes". This object is then identified in the analysis model editor (top right). In the navigator, the file VQPR005_FRM.java got a red dot. This means that this file contains a class that is mapped to the selected boundary object. When we open the file we can see highlighted all the methods that lead to the mapping. In the file editor, based on his knowledge of the implementation, the user can change the selection of the method signatures to complement the mapping done by the inference engine. The modified mapping will then be recorded by the system. For example, the user may know that some additional methods are involved in the implementation of a role of some class. This let the maintenance engineer work iteratively with the system when identifying the purpose of the implementation classes.

7. Conclusion and future work

In this paper we present the reverse-engineering tool we developed as a plugin to the eclipse environment. This plugin implements our approach about legacy software understanding. The first step is to identify the classes that implement a given use-case. This is relatively easy since we can locate them in the execution trace associated to the use-case. But this is not enough since there might be dozens of classes involved with many responsibilities. We need to know the role of these classes in the implementation of the use-case. Since we can design an analysis model for each use-case, we have a way to represent the roles of these classes. Therefore, if we can map the analysis object to the implementation classes, we get the role of the latter in the implementation. This is what our tool is able to do. However, since the implementation of a system can contain hundreds of classes and execution traces thousands if not millions of events, in general we cannot process this information by hand. Then we developed a mapping engine that is based on AI technologies. Our tool has been developed in Java as an Eclipse plugin. The experiments we have done so far on a medium size system (360 classes, 25'000 events in the trace) shows that the automatic matcher is able to get better results than the manual mapping. In fact its results are more precise than the ones we got by hand. This is because the automatic mapper is more systematic and processes all the information available in depth. For example, in this experiment, we also realized that we missed some mappings when processing the information by hand. Besides, our analyses with the tool also lead us to identify classes that played mixed roles. For example some classes played the role of a boundary and an entity object at the same time. This is usually the symptom of a bad design. Therefore, our tool could also be used to assess the quality of a design. As a next step in our research, we will extend our method and tool to let us compare the roles of the classes among all the use-cases of a software system. As a final remark, it is worth mentioning that our tool cannot "explain" (i.e. assign roles to) all the classes in the legacy system. In fact, some of the classes that represent exceptional situations or alternative execution paths cannot easily be identified since they might not be involved in the scenarios played by the users. Therefore, another step in our research will be to complement the tool with static analysis techniques to uncover the code that could potentially be executed in exceptional cases. Finally, we will also use domain ontologies to enhance the dynamic search for domain entitites in the programs.

8. Related work

Domain models have long been acknowledged as a good way to improve reverse engineering and program

understanding [14][15]. The authors usually propose a tool to support their approaches. The pioneering work can certainly be traced back to the famous RIGI system of Muller et al [12] that lead to the recent SHriMP & Creole systems [16]. Besides, DeBaud and Rugaber [5] and DeBaud [4] used an executable domain model in the form of an object oriented framework as the target of the understanding task. This framework represents the concept of the domain and helps the search for the corresponding concept in the programs. In the work of Gold [9], a knowledge base of programming concepts is used to help with the understanding problem. But these concepts are at a much lower level than the analysis model that we use. This approach is supported by the HB-CA tool. Rugaber and Stirewalt used a formal specification using an algebraic specification language to model both the domain and the program being reverse-engineered [14]. In the dynamic analysis approach to software understanding, many tools have been developed such as the work of Benett at al [3], Hamou-Lahdj [10], Zeidman et al [21]. There, the authors do not build higher conceptual models of the legacy system. Rather, the main concern is to cope with the quantity of information to display, to allow the maintenance engineer "understand" the involvement of the classes in the implementation of the system.

9. References

[1] Belmonte J., Dugerdil Ph. - Automating a domain model aware reengineering methodology. *Proc. Int. Conf. on Software Engineering and Knowledge Engineering,* 2008.

[2] Biggerstaff T.J., Mitbander B.G., Webster D.E. - Program Understanding and the Concept Assignment Problem", *Communications of the ACM, 37(5),* 1994.

[3] Bennett C., Myers D., Storey M.-A., German D. - Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer.*Proc. Workshop on Program Comprehension through Dynamic Analysis*, 2007.

[4] DeBaud J.-M. - Lessons from a Domain-Based Renegineering Effort. *Proc. IEEE Working Conf. on Reverse Engineering,* 1996.

[5] DeBaud J.-M., Rugaber S. - Software Reengineering method using Domain Models. *Proc. IEEE Int. Conf. on Software Maintenance,* 1995.

[6] Doyle J. - *A Truth Maintenance System. Artificial Intelligence 12*, *pp. 231-272*, 1979.

[7] Dugerdil Ph. - A Reengineering Process based on the Unified Process. *Proc. IEEE Int. Conf. on Software Maintenance,* 2006.

[8] Dugerdil Ph. - Using RUP to Reverse Engineer a Legacy System. *The Rational Edge,* September 2006.

[9] Gold N. - Hypothesis-Based Concept Assignment to Support Software Maintenance. *PhD Thesis, Univ. of Durham, UK*, 2000.

[10] Hamou-Lhadj A. - Towards a Multi-View Trace Visualization Environment. *Proc. of the 20th IEEE Canadian Conf. on Electrical and Computer Engineering,* 2007.

[11] Jacobson I., Booch G., Rumbaugh J. - *The Unified Software Development Process.* Addison-Wesley Professional, 1999.

[12] Müller H.A., Orgun M.A., Tilley S., Uhl J.S. - A Reverse Engineering Approach To Subsystem Structure Identification. *Software Maintenance: Research and Practice 5(4)*, John Wiley & Sons. 1993

[13] O'Brien M. - Software Comprehension – A Review & Research Direction. *Technical Report UL-CSIS-03-3, University of Limerick*, Nov. 2003

[14] Rugaber S., Stirewalt K. - Model-Driven Reverse Engineering. *IEEE Software*, July/August 2004.

[15] Sayyad-Shirabad J., Lethbridge T.C., Lyon S. - A Little Knowledge Can Go a Long Way Toward Program Understanding. *Proc IEEE Workshop on Program Comprehension,* 1997.

[16] Storey, M.-A., C. Best, J. Michaud, D. Rayside, M. Litoiu, M. Musen - "SHriMP views: an interactive environment for information visualization and navigation". *Proc. of the IEEE Int. Conf. on Human Factors in Computer Systems (CHI),* 2002.

[17] Storey M.-A. - Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Proc of the IEEE Int. Workshop on Program Comprehension,* 2005.

[18] Tilley S.R., Smith D.B., Paul S. - Towards a framework for program understanding. *Proc. IEEE Int. Workshop on Program Comprehension,* 1996.

[19] Horstmann C.S., de Pellegrin A. - Violet - http://sourceforge.net/projects/violet/

[20] von Mayrhauser A. – Program comprehension during software maintenance and evolution. *IEEE Computer 28(8)*, 1995.

[21] Zaidman A., Calders T., Demeyer S. Paredaens J. - Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process. *Proc. of the IEEE European Conf. on Software Maintenance and Reengineering ,* 2005.