Published in *International Journal of Software Engineering*
Vol. 2, N0. 3 December 2009, p. 135-150, which should be cited
to refer to this work

Using Robustness Diagrams to Help With Software Understanding: an Eclipse Plug-in

Philippe Dugerdil, Javier Belmonte, and David Kony

Department of Information Systems. HEG-Univ. of Applied Sciences (Switzerland)
E-mail: {philippe.dugerdil, 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 plug-in 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: Analysis tool, Dynamic analysis, Eclipse, Reverse engineering, Software understanding.

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 [1][2]. 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 [3], starts from the recovery of the system use-cases from its actual users. Its main steps are [2]:

- 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.

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. Fig. 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) [3].

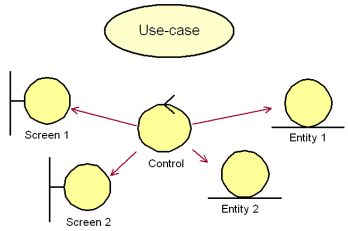

Figure 1 Use-case and analysis model

Besides, we must re-document the visible structure of the code based on syntactic clues such as module, package and class 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 (Fig. 2).

Figure 2 Use-case and the associated execution trace

The execution trace is obtained by instrumenting the source code of the legacy system. This allows us to trace the application classes only and to obtain an execution trace whatever the legacy programming language used. The trace format is the following:

[package][class][process][signature][returned type]

If the legacy language to be instrumented is not Java or C#, these concepts are mapped to the corresponding constructs of the language. [class] represents the entities in which the functions and procedures are declared. We may use UNITS in Cobol or Module in VB. [package] represent whatever structure that would group entities in larger chunks. [signature] is the signature of the function or procedure. [returned type] is the type of the object returned by the functions. With this format we always know the class or module whose methods or functions have been called. 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 [2]. The remaining classes are mapped to control role. 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. The first issue we encountered was that the involvement of analysis objects is never documented in association with the use-case themselves. Indeed, that's the case despite the fact that the main application of analysis models is the verification of the use-case's flow of events for business logic soundness [3]. We have then proposed to enhance the description of the use-case's flow of interactions in order to document such references. With this extension we must assign the analysis objects to each interaction they are involved in. This enhanced model can be referred as Enhanced Use Case Flows (EUCF). In Fig. 3 we present the UML model of such enhanced use-cases flows.

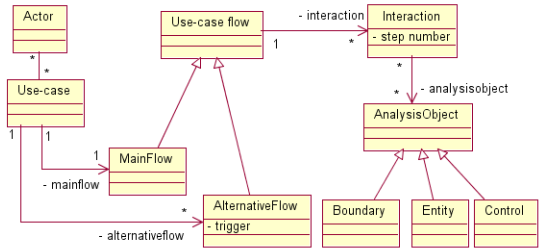

Figure 3 Model of the Enhanced Use-Case Flow (EUCF)

Once the EUCF is built, the sequence of involvement of the analysis objects during a use-case execution is straightforward. Fig. 4 presents a use-case with its main flow that is enhanced: to each of the steps (interaction) the involved analysis objects are referenced.

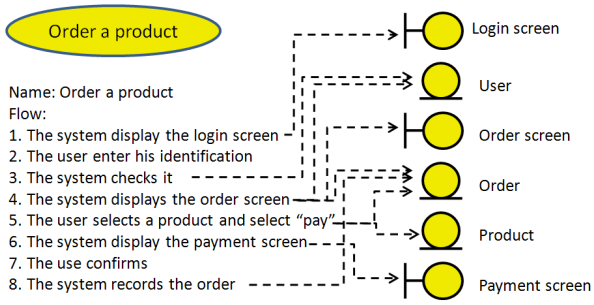

Figure 4 Use case and involved objects

By comparing the sequence of analysis object involvements in the use-case flow with the sequence of implementation classes' occurrences in the corresponding execution trace, we can map the objects as showed in Fig. 5.

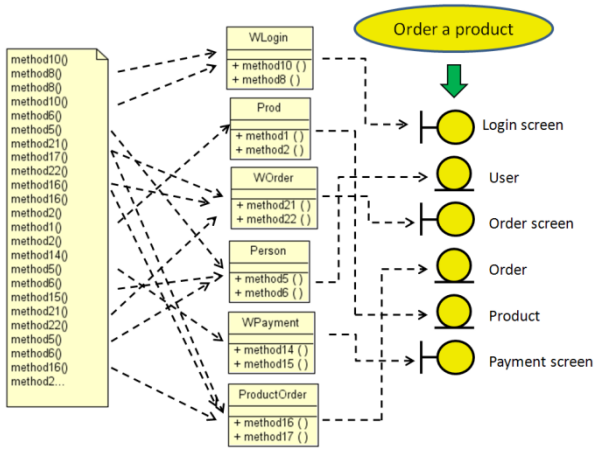

Figure 5 Mapping implementation classes to analysis object based on sequence

To strengthen the mapping between the classes of the analysis model and the implementation model, we also compare the associations in the analysis model to the ones in the implementation classes as showed in Fig. 6.

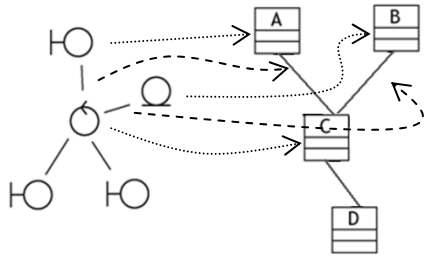

Figure 6 Comparison of association

Once the first results were obtained, it became clear that having a single control object per use-case as suggested by the Unified Process, was too coarse grained for us to end up with a useful mapping for control objects. We therefore tried out another extension, this time on the process itself, to allow for more than a single control object per use-case. Each of them would now bear fewer responsibilities. The final set of interacting control objects would then implement the global coordination of the use-case. Finally, 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.

3- SOFTWARE UNDERSTANDING JUSTIFICATION

Software understanding theories have long been reported in the literature [4-9]. 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 may be involved in the implementation of several analysis objects. Moreover, several classes may implement a single analysis object. 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, the environment is also required 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 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 [4]. 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 [10] 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.

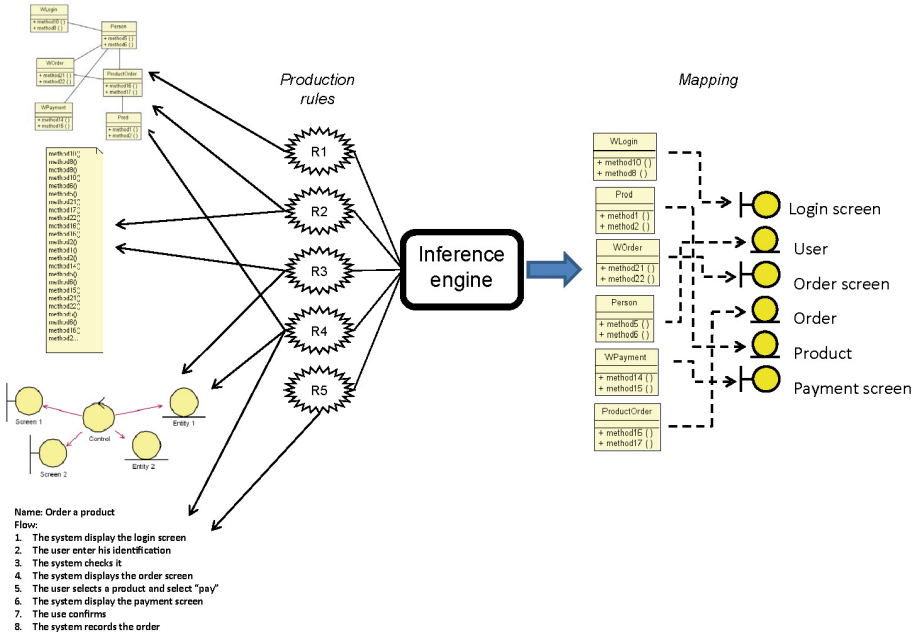

Figure 7 Inference engine to infer the mapping

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 Fig. 7. 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 analysis model;
- The methods in the implementation classes that lead to 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 PLUG-IN

Our tool, which is developed as a plug-in to the Eclipse platform[1], 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.

Fig. 8 presents the integrated reverse engineering environment in the Eclipse

[1] Since our tool has been developed as an independent Eclipse plug-in, it could be loaded into IBM's Rational Software Architect.

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 analysis model editor. This tool is an open source plug-in 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 the most suitable to be Violet [11]. The use-case editor is represented on the bottom. It has three subviews. On the left one it 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 interaction step in the use-case flow. The objects represented on the bottom right are the one associated to the 6th interaction step in the use-case flow. The column “stat” displays 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.

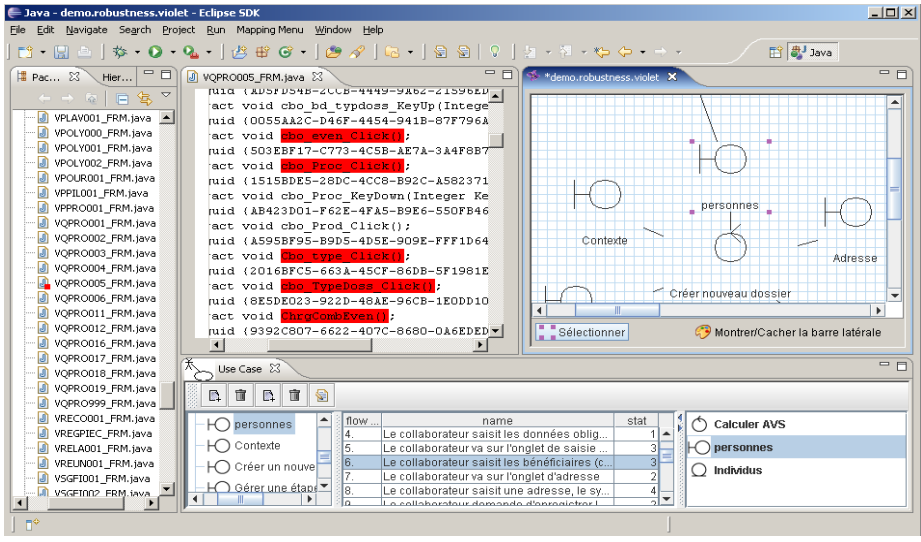

Figure 8 The reverse engineering environment under Eclipse

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 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 Fig. 8) 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 Fig. 8, 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- MAPPING RESULTS

To validate our approach, we must compare the mapping performed by hand with the mapping performed automatically with the help of our tool. In the discussion below, we will separately discuss the most interesting case: the mapping of the boundary and the control objects. As for the entity objects, the mapping are more straightforward since we analyze database accesses. Therefore the mapping is always very close.

7-1 RESULTS IN MAPPING BOUNDARY OBJECTS

Since the set of rules that we implemented were inspired by a previous manual application of the methodology [12], we thought on comparing their results on the same problem instance. Table 1 shows the results obtained by both approaches; highlighted modules are those that were mapped differently. Our system was therefore able to correctly establish 81% of the manual mappings and at the same time 93% of the mappings automatically made were correct. In both cases, the ratio is in favor of the automatic mapping.

TABLE 1 COMPARISON OF THE MANUAL AND AUTOMATIC MAPPINGS

Boundary object	Manual map	Automatic map
Create a new folder	VQPRO005.FRM	VQPRO005.FRM
Context	VQPRO005.FRM	VQPRO005.FRM
Persons	VQPRO005.FRM Z_RGEN00.FRM VINDI001.FRM	VQPRO005.FRM
Address	VQPRO005.FRM Z_DGEN01.FRM	VQPRO005.FRM Z_DGEN01.FRM
Address input	VQPRO005.FRM Z_DGEN01.FRM	VQPRO005.FRM Z_DGEN01.FRM
Folder explorer	VQPRO004.FRM	VQPRO004.FRM
Step management	FMENUPOP.FRM	FMENUPOP.FRM
Evaluation	VXTRT004.FRM	VXTRT004.FRM Z_ATTENT.FRM
Modalities	VNINT001.FRM	VNINT001.FRM
Intervention decision	VNINT001.FRM	VNINT001.FRM
Characteristics	VNINT001.FRM Z_ATTENT.FRM	VNINT001.FRM

7-2 RESULTS IN MAPPING CONTROL OBJECTS

In the initial work performed by hand, we stick to the UP heuristic to associate a single control object per use-case. Therefore the comparison with the automatic match is difficult since, as presented above, we now allow for several control objects to be associated to a single use-case. Nevertheless, we can analyze the process and see if the results agree with what can be manually observed. The resulting mapping is detailed in Table 2.

TABLE 2 LIST OF CONTROL OBJECT AND THE CLASSES THEY WERE MAPPED TO

Class	Analysis object	Certainty
X_SPREAD	Control-01	0.53
X_SPREAD	Control-06	0.49
Z_SERVIC	Control-01	0.56
Z_DGEN01.FRM	Control-02	0.54

The heuristic behind this mapping works on the interaction among classes that are already mapped to analysis objects [13]. As an example of such a heuristics, let us assume that classes A, B an C have been mapped to the entity and boundary objects around the control object (Fig. 9) We can then infer the mapping between C and that control object because of the relationship between the objects in both models. Specifically, the black lines between the classes represent dynamic interactions: the thicker the line the greater the

correlation between those classes during the analyzed execution.

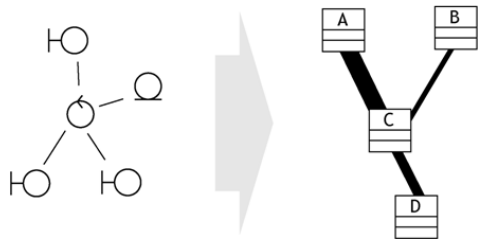

Figure 9 The structural match heuristic

We present in Fig. 10 the results of the mapping summarized in Table 2. The graph on the top is a partial view of the analysis model of the use-case. The graph on the bottom is a view of the dynamic interaction among the classes as recovered from the execution trace associated to the use-case. We only presented the most important dependencies among the five classes already matched. Gray levels between the classes and the analysis objects in the figure correspond to the mapping in Table 2.

The matching of the classes *Z_SERVIC* and *X_SPREAD* seems correct since both classes are associated with the classes implementing the boundary objects of the analysis diagram substructure. The matching of the class *Z_DGEN01.FRM* to a control object is an interesting case. It highlights some weak semantic coherence in the legacy program since this class has also been found to implement a boundary object in section 7-1. Since we keep track of the methods leading to each match, it is easy to differentiate between the different roles played by a single implementation class. Then, we could selectively highlight the methods belonging to each role. In the future we could use this information to help with the refactoring of the implementation classes into single-role classes.

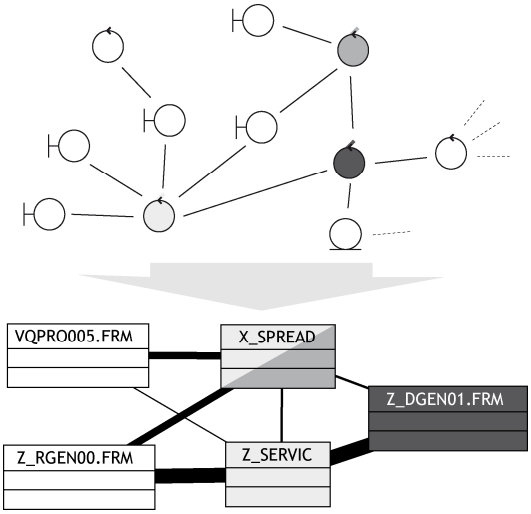

Figure 10 Robustness diagram and corresponding implementation classes

8- CONCLUSION AND FUTURE WORK

In this paper we present the reverse-engineering tool we developed as a plug-in to the eclipse environment. This plug-in 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 plug-in. 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 mapping program 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.

Much of what has been achieved with our system is possible because of the re-documentation step at the beginning of the methodology, and especially the analysis models. In fact, the latter is most of the times considered only as a help to the forward development process and, sometimes, to the (partial) code generation automation. Our results have highlighted their invaluable relevance to the reverse-engineering process and the automation of program understanding. We hope to encourage development teams to create analysis models since, beyond being of a great help in the initial development process, they will keep their usefulness over longer maintenance periods since they are much less volatile than design models.

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 use domain ontologies to enhance the dynamic search for domain entities in the programs.

9- RELATED WORK

Domain models have long been acknowledged as a good way to improve reverse engineering and program understanding [7][14]. 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 [15] that lead to the recent SHriMP & Creole systems [16]. Besides, DeBaud and Rugaber [17] and DeBaud [18] 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 [19], 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 [7].

In the dynamic analysis approach to software understanding, many tools have been developed such as the work of Benett at al [20], Hamou-Lahdj [21], Zeidman et al [22]. 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. Other, such as Quante J. et al. [23] or Smit M. et al. [24] try to recover formal protocols about program behavior, that could later be transformed into something very close to a use-case description. Even though our approach works the other way around, the lessons learned therein could help extending our methodology to discover new use-cases that can’t be foreseen by the users, or very particular execution paths in known use-cases.

In what concerns the reverse mapping of implementation classes to the analysis objects, we couldn’t find any approaches with whom to compare our work. This might be a consequence of the analysis model being only very seldom referenced in the reverse engineering literature.

REFERENCES

[1] Ph. Dugerdil, “A Reengineering Process based on the Unified Process,” Proc. IEEE Int. Conf. on Software Maintenance, 2006.

[2] Ph. Dugerdil, “Using RUP to Reverse Engineer a Legacy System,” The Rational Edge, September 2006.

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

[4] J. Belmonte, Ph. Dugerdil, “Automating a domain model aware reengineering methodology,” Proc. Twentieth Int. Conf. on Software Engineering and Knowledge Engineering, 2008.

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

[6] M. O’Brien, “Software Comprehension – A Review & Research Direction,” Technical Report UL-CSIS-03-3, University of Limerick, Nov. 2003.

[7] S. Rugaber, K. Stirewalt, “Model-Driven Reverse Engineering,” IEEE Software, July/August 2004.

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

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

[10] J. Doyle, “A Truth Maintenance System,” Artificial Intelligence, vol. 12, pp. 231-272, 1979.

[11] C.S. Horstmann, A. de Pellegrin, “Violet,” Violet, http://sourceforge.net/projects/violet/. 2008.

[12] S. Jossi, “Reverse-engineering du système d’information,” Rapport de travail de Diplôme, Haute École de Gestion de Genève (HEG-GE), 2006.

[13] J. Belmonte, “Automatisation d’une méthode de Reverse Engineering basée sur un système de production,” Master’s degree research final report (University of Geneva – HEG-GE), 2008.

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

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

[16] M.-A. Storey, 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] J.-M. DeBaud, S. Rugaber, “Software Reengineering method using Domain Models,” Proc. IEEE Int. Conf. on Software Maintenance, 1995.

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

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

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

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

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

[23] J. Quante, R. Koschke, “Dynamic Protocol Recovery,” Proc. 14th Working Conf. on Reverse Engineering, 2007.

[24] M. Smit, E. Stroulia, K. Wong, “Use Case Redocumentation from GUI Event Traces,” Proc. 12th European Conf. on Software Maintenance and Reengineering, 2008.