ROLE-BASED CLUSTERING OF SOFTWARE MODULES
An Industrial Size Experiment

Philippe Dugerdil and Sebastien Jossi
Department of Information Systems, HEG-Univ. of Applied Sciences, 7 rte de Drize, 1227 Geneva, Switzerland

Keywords: Reverse-engineering, software process, software clustering, software reengineering, program comprehension, industrial experience.

Abstract: Legacy software system reverse engineering has been a hot topic for more than a decade. One of the key problems is to recover the architecture of the system i.e. its components and the communications between them. Generally, the code alone does not provide much clue on the structure of the system. To recover this architecture, we proposed to use the artefacts and activities of the Unified Process to guide the search. In our approach we first recover the high-level specification of the program. Then we instrument the code and "run" the use-cases. Next we analyse the execution trace and rebuild the run-time architecture of the program. This is done by clustering the modules based on the supported use-case and their roles in the software. In this paper we present an industrial validation of this reverse-engineering process. First we give a summary of our methodology. Then we show a step-by-step application of this technique to real-world business software and the result we obtained. Finally we present the workflow of the tools we used and implemented to perform this experiment. We conclude by giving the future directions of this research.

1 INTRODUCTION

To extend the life of a legacy system, to manage its complexity and decrease its maintenance cost, it must be reengineered. However, reengineering initiatives that do not target the architectural level are more likely to fail (Bergey et al. 1999). Consequently, many reengineering initiatives begin by reverse architecting the legacy software. The trouble is that, usually, the source code does not contain many clues on the high level components of the system (Kazman, O'Brien, Verhoef 2003). However, it is known that to "understand" a large software system, which is a critical task in reengineering, the structural aspects of the software system i.e. its architecture are more important than any single algorithmic component (Tilley, Santanu, Smith 1996). A good architecture is one that allows the observer to "understand" the software. To give a precise meaning to the word "understanding" in the context of reverse-architecting, we borrow the definition by Biggerstaff et al. (Biggerstaff, Mitbander, Webster 1994): "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program".

In other words, the structure of the system should be mappable to the domain concepts (what is usually called the "concept assignment problem"). In the literature, many techniques have been proposed to split a system into components. These techniques range from clustering (Andritsos, Tzerpos 2005) (Wen, Tzerpos 2004), slicing (Verbaere 2003) to the more recent concept analysis techniques (Eisenbarth, Koschke 2003)(Tonella 2001) or even mixed techniques (Tonella 2003). However the syntactical analysis of the mere source code of a system may produce clusters of program elements that cannot be easily mapped to domain concepts because both the domain knowledge and the program clusters have very different structures. However to find a good clustering of the program elements (i.e. one for which the concept assignment is straightforward) one should first understand the program. But to understand a large software system one should know its structure. This resembles the chicken and egg syndrome. To escape from this situation, we propose to start from an hypothesis on the architecture of the system. Then we proceed with the validation of this

Dugerdil P. and Jossi S. (2007).
ROLE-BASED CLUSTERING OF SOFTWARE MODULES - An Industrial Size Experiment.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 5-12
DOI: 10.5220/0001329100050012
Copyright © SciTePress

architecture using a run time analysis of the system. The theoretical framework of our technique has been presented elsewhere (Dugerdil 2006). In this paper we present the result of the reverse engineering of an industrial-size legacy system. This shows that our technique scales well and allows the maintenance engineer to easily map high-level domain concepts to source code elements. It then helps him to “understand” the code, according to the definition of Biggerstaff et al.

2 SHORT SUMMARY OF OUR METHOD

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 that 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 technique starts from the recovery of the system use-cases from its actual users and proceeds with following steps (Dugerdil 2006):

- Redocumentation of the system use-cases;
- Redocumentation of the corresponding business model;
- Design of the robustness diagram associated to all the use-cases;
- Redocumentation of the high level 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 trace to the classes of the robustness diagram with analysis of the roles.
- Redocumentation of the architecture of the system by clustering the modules based on their role in the use-case implementation.

Figure 1 shows a use-case model and the corresponding business analysis model. Then, for each use-case we rebuild the associated robustness diagram (UML2 name for the Analysis Model of the Unified Development Process (Jacobson, Booch, Rumbaugh 1999)). These robustness diagrams represent our best hypothesis on the actual architecture of the system. Then, in the subsequent steps, we must validate this hypothesis and identify the roles the modules play. Figure 2 presents an example of a robustness diagram with their UML stereotypical classes that represent software roles for the classes (Jacobson, Booch, Rumbaugh 1999).

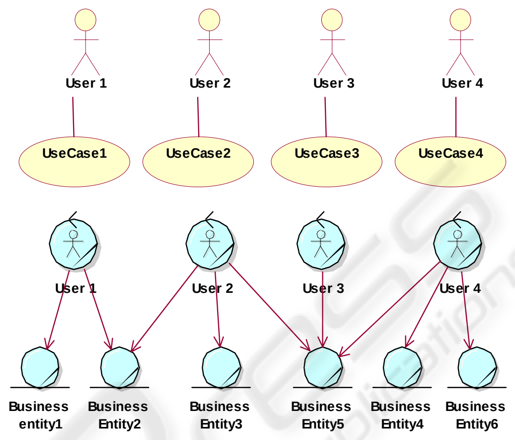

Figure 1: Use-case model and business model.

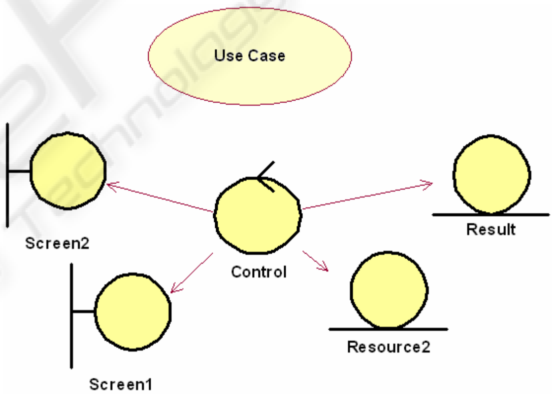

Figure 2: Use-case and robustness diagram.

The next step is to recover the visible high level structure of the system (classes, modules, packages, subsystems) from the analysis of the source code, using the available syntactic information (fig 3).

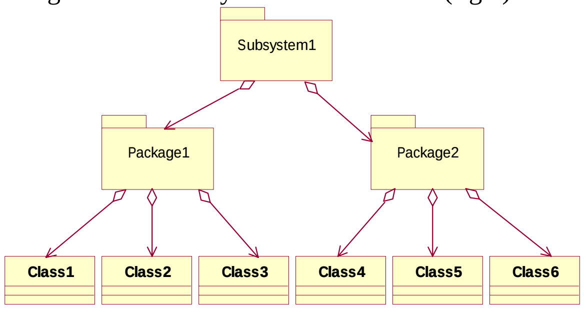

Figure 3: The high-level structure of the code.

Now, we must validate our hypothetical architecture (robustness diagrams) against the actual code of the system and find the mapping from the stereotypical classes to the actual modules. First, we run the system according to each use-case and record the execution trace (fig. 4).

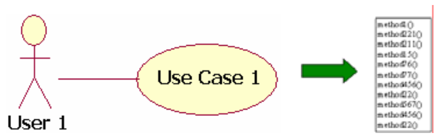

Figure 4: Use-case and the associated execution trace.

Next, the functions in the trace are linked to the classes or modules they belong to. These are the classes or modules that actually implement the use-case. These classes or modules are then highlighted in the high level structure of the code (fig. 5).

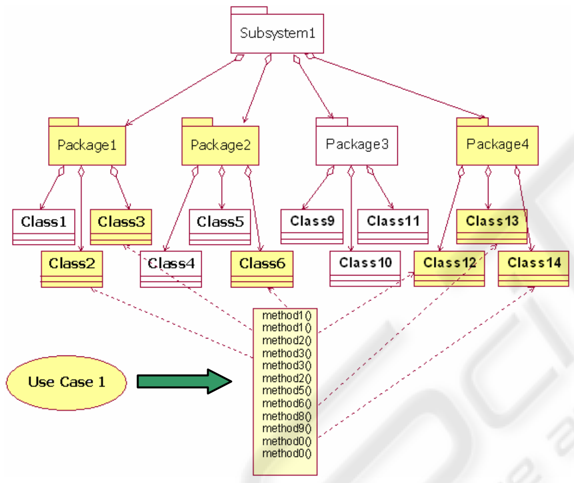

Figure 5: From the trace to the high level structure of the code.

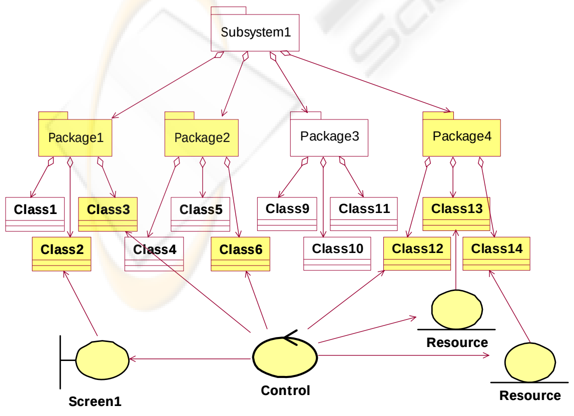

Figure 6: Mapping actual classes to software roles.

The classes found are further analysed to find evidence of a database access function or of a screen display function. This let us categorize the classes as entities (access to database tables) or boundaries (interface to the user). The remaining classes will be categorized as control classes. Figure 6 presents the result of such a mapping. The last step in our method is to cluster the actual classes or modules according to the use-case they implement and to their role as defined above. This represents the recovered architecture of the system. Figure 7 shows such a recovered architecture for a single use-case.

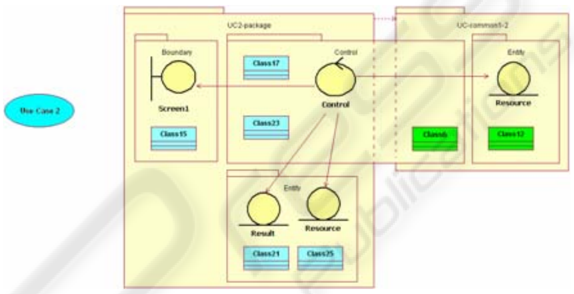

Figure 7: Recovered architecture for a use-case.

3 INDUSTRIAL EXPERIMENT

This technique has been applied to an industrial packaged software. This system manages the welfare benefit in Geneva. It is a fat-client kind of client-server system. The client is made of 240k lines of VB6 code. The server consists of 80k lines of PL/SQL code accessing an Oracle database. In this paper, for the sake of conciseness, we will concentrate on the reverse engineering of the client part of this system. But the technique has been applied as well to the server part.

3.1 Recovering the Use-cases

Due to heavy workload of the actual users of this system we recovered the used-cases by interacting with the user-support people who know the domain tasks perfectly well. Then we documented the 4 main use-cases of the system by writing down the user manipulation of the system and video recording the screens through which the user interacted (Figure 8). From this input, we were able to rebuild the business model of the system (Figure 9). In the latter diagram, we show the workers (the tasks) and the resources used by the workers. Each system actor of the use-case model corresponds to a unique worker in the business model. The technique used to infer the business model come from the Unified Process.

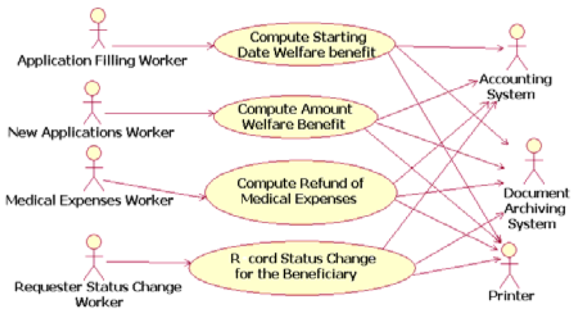
Application Filling Worker
New Applications Worker
Medical Expenses Worker
Requester Status Change Worker
Compute Starting Date Welfare benefit
Compute Amount Welfare Benefit
Compute Refund of Medical Expenses
R cord Status Change for the Beneficiary
Accounting System
Document Archiving System
Printer

Figure 8: The main use-cases of the system.

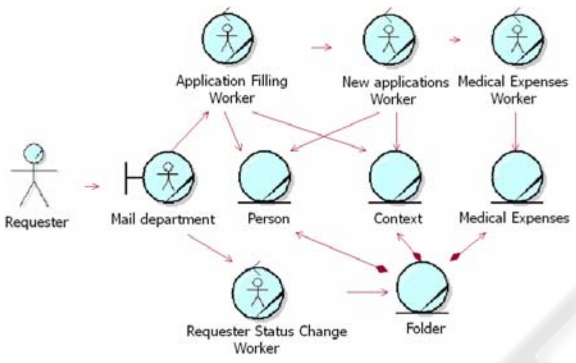

Figure 9: The recovered high level business model.

3.2 Recovering the Visible Structure

In our experiment we did not have any specific tool at our disposal to draw the modules and module dependencies, neither for VB6 nor PL/SQL. However, we regularly use the Rational/IBM XDE environment which can reverse-engineer Java code.

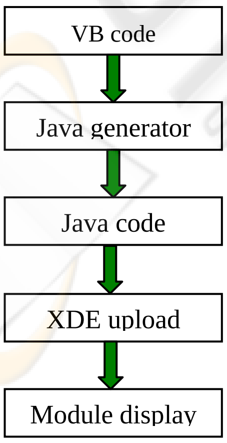

Figure 10: Visible high level structure extraction workflow.

Then, we decided to generate skeleton Java classes from both the VB6 and PL/SQL code to benefit from XDE. Each module in VB6 or PL/SQL is represented as a class and the dependencies between modules as associations. We then wrote in Java our own VB6 parser and Java skeleton generator. Figure 10 presents the workflow for the display of the visible high-level structure of the VB6 client tier of the system. The resulting high-level structure diagram is presented in figure 11. There are 360 modules in this diagram.

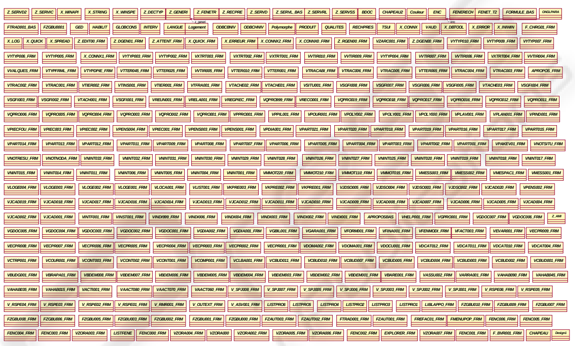

Figure 11: Visible high level structure of the client.

3.3 Building the Robustness Diagram

The robustness diagram is built by hand using the heuristics set forth by the Unified Process. Figure 12 presents the robustness diagram of the first use-case called “Compute the starting date of welfare benefit”. It is the second largest use-case of this system. Again, this diagram has been built from the analysis of the use-case only, without taking the actual code into account.

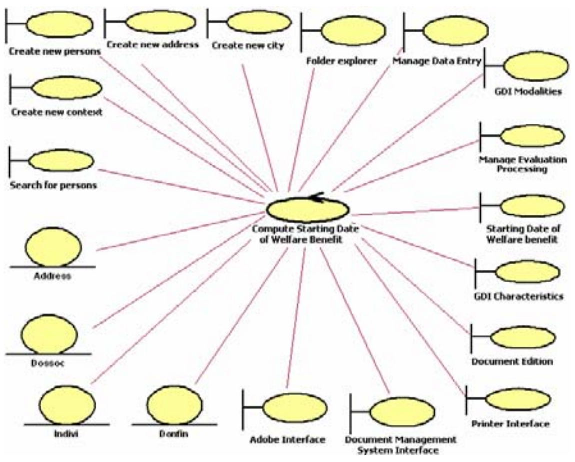

Figure 12: Robustness diagram of the first use-case.

3.4 Running the Use-case

The next step is to “execute” the use-cases i.e. to run the system following the manipulations expressed by the use-cases. Then the execution trace must be recorded. Again, we have not found any specific environment able to generate an execution trace for the client part written in VB6. Then we decided to instrument the code to generate the trace (i.e. insert trace generation statement in the source code). Therefore we wrote an ad-hoc VB6 instrumentor in Java. The modified VB6 source code must then be recompiled before being executed. The format of the trace we generate is:

```
<moduleName><functionSignature><parameterValues>
```

The only parameter values we record in the trace are the one with primitive types, because we are interested in SQL statements passed as parameters. This will help us find the modules playing the role of “Entities” (see below). For the server part (PL/SQL), the trace can be generated using the system tools of Oracle.

3.5 Trace Analysis

In the next step we analysed the trace to find the modules involved in the execution. The result for the client part is presented in figure 13. We found that only 44 modules are involved in the processing of this use-case, which is one of the biggest in the application. But this should not come as a surprise. Since this system is a packaged software, then a lot of the implemented functions are unused.

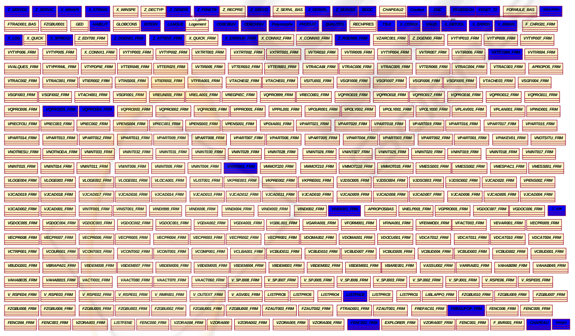

Figure 13: Modules involved in the first use-case.

The last step is to sort out the roles of the modules in the execution of the use case. This will allow us to cluster the modules according to their role. Then we analysed the code of the executed functions to identify screen-related functions (i.e. VB6 functions used to display information). The associated modules then play the role of the boundaries in the robustness diagram. Next, we analysed the parameter values of the functions to find SQL statements. The corresponding modules play the role of the entities in the robustness diagram. The remaining modules play the role of the control object. The result of this analysis is presented in figure 14. The modules in the top layer (red) are boundaries (screens), the bottom layer (yellow) are the entities and the middle layer (blue) contains the modules playing the role of the control object.

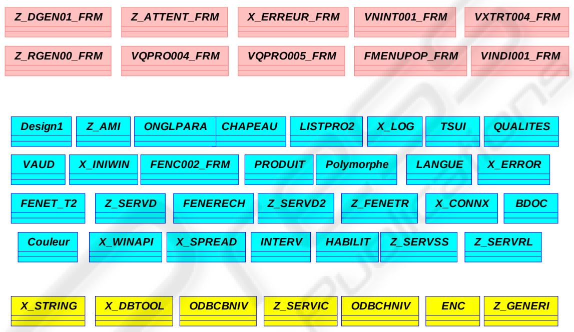

Figure 14: software roles of the involved modules.

As an alternative view, we map the modules of the client part to the robustness diagram we built from the use-case. By correlating the sequence of use in the use-case and the sequence of appearance in the execution we can identify each boundary object. As for the entity objects, they are not specific to any given table. In fact, we found that each entity module is involved in the processing of many tables.

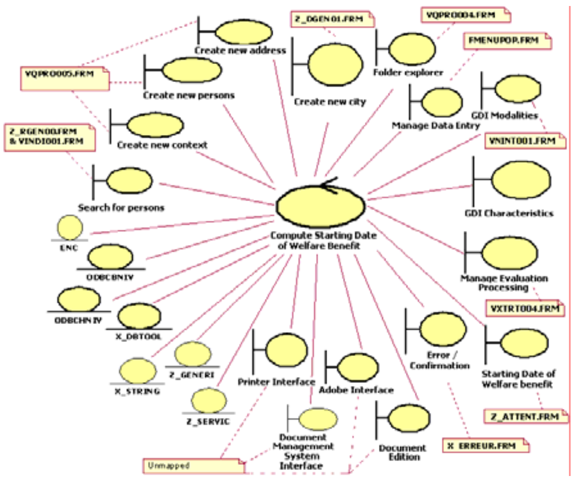

Figure 15: Modules to robustness diagram mapping.

Then, we represented all of them in the robustness diagram without mapping to any specific database table. The result of the mapping is represented in figure 15. Since the number of modules that play the role of the control object is large, they are not shown in the diagram. In this experiment, we were not able to map the boundary labelled with the “unmapped” note (bottom). In fact they represent interfaces with external systems that we cannot reach from the test environment we used in our experiments. Therefore no mapping was possible.

3.6 Role-based Clustering of Modules

Figure 16 represents the role-based clustering of the client modules identified in our experiment. First, all the modules are grouped in a package named after the use-case they implement. Second the modules are grouped after the Robustness-Diagram role (§2) they play in this implementation. This represents a specific view of the system’s architecture. It corresponds to the role the module play in the currently implemented system. It is important to note that this architecture can be recovered whatever the maintenances to the system. Since it comes from the execution of the system, it can cope with the dynamic invocation of modules, something particularly difficult to analyse using static analysis only.

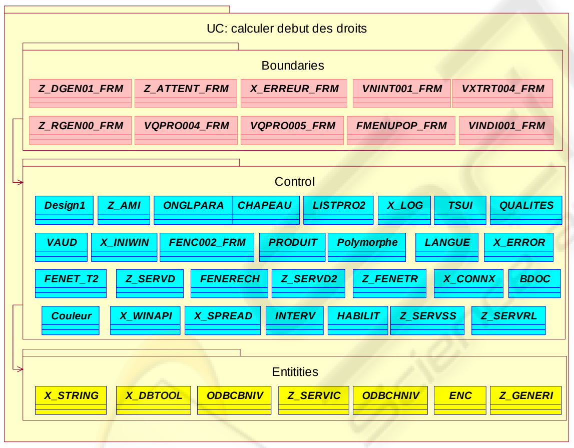

Figure 16: Recovered architecture of the main use-case of the system.

4 TOOLS WORKFLOW

In figure 17, we present the overall workflow of the tools we used to analyse the system. On the left we find the tools to recover the visible high level structure of the system. On the right we show the tools to generate and analyse (filter) the trace. In the center of the figure we show the use-case and the associated robustness diagram.

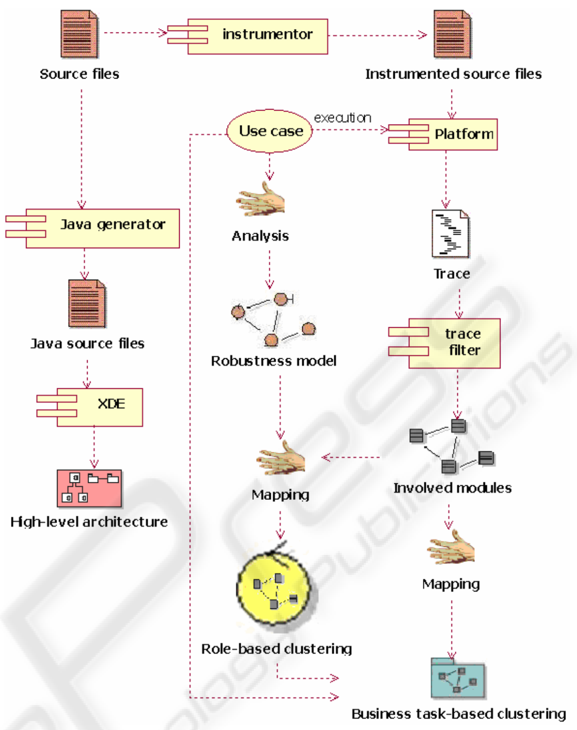

Figure 17: Workflow of the reverse-engineering tools.

5 RELATED WORK

The problem to link the high level behaviour of the program to the low-level software components has been the source of many research works and publications. Often, in the literature, the authors try to solve the problem by designing an algorithm that groups the software elements according to some criteria. Among the most popular techniques we find static clustering and formal concept analysis.

- The clustering algorithms groups the statements of a program based on the dependencies between the elements at the source level, as well as the analysis of the cohesion and coupling among candidate components (Mitchell 2003) (Kuhn, Ducasse, Girba 2005).
- Formal concept analysis is a data analysis technique based on a mathematical approach to group the «objects» that share some common «attributes». Here the object and attributes can be any relevant software elements. For example, the objects can be the program functions and the attributes the variables accessed by the functions (Linding, Snelting 1997) (Siff, Reps 1999). For

example, this technique has been proposed to identify the program elements associated to the visible features of the programs (Rajlich, Wilde 2002) (Eisenbarth, Koschke 2003).

In fact, these techniques try to partition the set of source code statements and program elements into subsets that will hopefully help to rebuild the architecture of the system. The key problem is to choose the relevant set of criteria (or similarity metrics (Wiggert 1997)) with which the “natural” boundaries of components can be found. In the reverse-engineering literature, the similarity metrics range from the interconnection strength of Rigi (Müller, Orgun, Tilley, Uhl 1993) to the sophisticated information-theory based measurement of Andritsos (Andritsos, Tzerpos 2003) (Andritsos, Tzerpos 2005), the information retrieval technique such as Latent Semantic Indexing (Marcus 2004) (Kuhn, Ducasse, Girba 2005) or the *kind* of variables accessed in formal concept analysis (Siff, Reps 1999) (Tonella 2001). Then, based on such a similarity metric, an algorithm decides what element should be part of the same cluster (Mitchell 2003). On the other hand, Gold proposed a concept assignment technique based on a knowledge base of programming concepts and syntactic “indicators” (Gold 2000). Then, the indicators are searched in the source code using neural network techniques and, when found, the associated concept is linked to the corresponding code. However he did not use his technique with a knowledge base of domain (business) concepts. In contrast with these techniques, our approach is “business-function-driven” i.e. we clusters the software elements according to the supported the business tasks and functions. The domain modelling discipline of our reverse-engineering method presents some similarity with the work of Gall et al. (Gall, Klosch, Mittermeier 1996) (Gall, Weidl 1999). These authors tried to build an object-oriented representation of a procedural legacy system by building two object models. First, with the help of a domain expert, they build an abstract object model from the specification of the legacy system. Second, they reconstruct an object model of the source code, starting from the recovered entity-relationship model to which they append dynamic services. Finally, they try to match both object models to produce the final object oriented model of the procedural system. The authors report that one of the main difficulties is the assignation of the dynamic features to the recovered objects (what they call the “ambiguous service candidates”). In contrast, our approach does not try to transform the legacy system into some object-oriented form. The robustness diagram we build is simply a way to document the software roles. Our work bears some resemblance to the work of Eisenbarth and Koschke (Eisenbarth, Koschke 2003) who used Formal Concept Analysis. However the main differences are:

1. The scenarios we use have a strong business-related meaning rather than being built only to exhibit some features. They represent full use-cases.
2. The software clusters we build are interpretable in the business model. We do group the software element after their roles in the implementation of business functions.
3. We analyse the full execution trace from a real-use-case to recover the architecture of the system.
4. The elements we cluster are modules or classes identified in the visible high-level structure of the code.

Finally, it is worth noting that the use-cases play, in our work, the same role as the test cases in the execution slicing approach of Wong et al. (Wong, Gokhale, Horgan, Trivedi 1999). However, in our work, the “test cases” are not arbitrary but represent actual use-cases of the system.

6 CONCLUSION

The reverse-engineering process we present in this article rests on the Unified Process from which we borrowed some activities and artefacts. The techniques are based on the actual working of the code in real business situations. Then, the architecture we end up with is independent on the number of maintenances to the code. Moreover it can cope with situation like dynamic calls, which are tricky to analyse using static techniques. We actually reverse-engineered all the use-cases of the system and found that the modules involved in all of them were almost the same. Finally, this experiment seems to show that this technique is scalable and is able to deal with industrial size software.

As a next step in this research we are developing a semi-automatic robustness diagram mapper that takes a robustness diagram and a trace file as input and produces a possible match as output. This system uses a heuristic-based search engine coupled to a truth maintenance system.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the "Reserve Strategique" of the Swiss Confederation (Grant ISNET 15989). We also thank the people at the CTI of the Canton Geneva (Switzerland) who helped us perform the industrial experiment.

REFERENCES

Andritsos P., Tzerpos V. 2003. *Software Clustering based on Information Loss Minimization*. Proc. IEEE Working Conference on Reverse engineering.

Andritsos P., Tzerpos V. 2005. Information Theoretic Software Clustering. IEEE Trans. on Software Engineering 31(2), 2005.

Bergey J. et al. 1999. *Why Reengineering Projects Fail*. Software Engineering Institute, Tech Report CMU/SEI-99-TR-010, Apr. 1999.

Bergey J., Smith D., Weiderman N., Woods S. 1999. *Options Analysis for Reengineering (OAR): Issues and Conceptual Approach*. Software Engineering Institute, Tech. Note CMU/SEI-99-TN-014.

Biggerstaff T. J., Mitbander B.G., Webster D.E. 1994. *Program Understanding and the Concept Assignment Problem*. Communicaitons of the ACM, CACM 37(5).

Binkley D.W., Gallagher K.B. 1996. *Program Slicing*. Book chapter in: Advances in Computers, vol 43, Academic Press, 1996.

Dugerdil Ph. 2006. *A Reengineering Process based on the Unified Process*. IEEE International Conference on Software Maintenance.

Eisenbarth T., Koschke R. 2003. *Locating Features in Source Code*. IEEE Trans. On Software Engineering 29(3) March 2003.

Gall H., Klosch R. Mittermeir R. 1996. *Using Domain Knowledge to Improve Reverse Engineering*. Int. J. on Software Engineering and Knowledge Engineering (IJSEKE), 6(3).

Gall H., Weidl J. 1999. *Object-Model Driven Abstraction to Code Mapping*. Proc. European Software engineering Conference, Workshop on Object-Oriented Reengineering.

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

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

Kazman R., O'Brien L., Verhoef C. 2003. *Architecture Reconstruction Guidelines*, 3rd edition. Software Engineering Institute, Tech. Report CMU/SEI-2002-TR-034.

Kuhn A., Ducasse S., Girba T. 2005. *Enriching Reverse Engineering with Semantic Clustering*. Proc. IEEE IEEE Working Conference on Reverse engineering.

Linding C., Snelting G. 1997. *Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis*. Proc IEEE Int. Conference on Software Engineering .

Marcus A. 2004. Semantic Driven Program Analysis. Proc IEEE Int. Conference on Software Maintenance .

Mitchell B.S. 2003. A Heuristic Search Approach to Solving the Software Clustering Problem. Proc IEEE Conf on Software Maintenance.

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

Wiggert T.A. 1997 – *Using Clustering Algorithms in Legacy Systems Remodularisation*. Proc. IEEE Working Conference on Reverse engineering.

Rajlich V., Wilde N, 2002. *The Role of Concepts in Program Comprehension*. Proc IEEE Int. Workshop on Program Comprehension.

Siff M., Reps T. 1999. *Identifying Modules via Concept Analysis*. IEEE Trans. On Software Engineering 25(6).

Tilley S.R., Santanu P., Smith D.B. 1996. Toward a Framework for Program Understanding. Proc. IEEE Int. Workshop on Program Comprehension.

Tonella P. 2001. *Concept Analysis for Module Restructuring*. IEEE Trans. On Software Engineering, 27(4).

Tonella P. 2003. *Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis*. IEEE Trans. On Software Engineering. 29(6)

Verbaere M. 2003 - *Program Slicing for Refactoring*. MS Thesis, Oxford University.

Wen Z., Tzerpos V. 2004 – *An Effective measure for software clustering algorithms*. Proc IEEE Int. Workshop on Program Comprehension.

Wiggert T.A. 1997. *Using Clustering Algorithms in Legacy Systems Remodularisation*. Proc. IEEE Working Conference on Reverse engineering.

Wong W.E., Gokhale S.S., Horgan J.R., Trivedi K.S. 1999. *Locating Program Features using Execution Slices*. Proc. IEEE Conf. on Application-Specific Systems and Software Engineering & Technology.