FAMOOSr 2007

1st Workshop on FAMIX and Moose in Reengineering

Organizers:

Tudor Gîrba	University of Bern, Switzerland
Adrian Kuhn	University of Bern, Switzerland

Program Committee:

Juan-Carlos Cruz	UBS, Switzerland
Serge Demeyer	University of Antwerp, Belgium
Orla Greevy	University of Bern, Switzerland
Michele Lanza	University of Lugano, Switzerland
Radu Marinescu	Politehnica University of Timisoara, Romania
Kim Mens	Université catholique de Louvain, Belgium
Tom Mens	University of Mons-Hainaut, Belgium
Oscar Nierstrasz	University of Bern, Switzerland
Martin Pinzger	University of Zurich, Switzerland
Sander Tichelaar	Credit Suisse, Switzerland
Roel Wuyts	Université Libre de Bruxelles, Belgium

About FAMOOSr

The goal of the FAMOOSr workshop is to strengthen the community of researchers and practitioners who are working in re- and reverse engineering, by providing a forum for building future research using Moose and FAMIX as shared infrastructure.

Research should be collaborative and supported by tools. The increasing amount of data available about software systems poses new challenges for reengineering research, as the proposed approaches need to scale. In this context, concerns about meta-modeling and analysis techniques need to be augmented by technical concerns about how to reuse and how to build upon the efforts of previous research.

That is why Moose is an open-source software for researchers to build and share their analysis, meta-models, and data. Both FAMIX and Moose started in the context of FAMOOS, a European research project on object-oriented frameworks. Back in 1997 Moose was as a simple implementation of the FAMIX meta-model, which was a language independent meta-model for object-oriented systems. However over the past decade, Moose has been used in a growing number of research projects and has evolved to be a generic environment for various reverse and reengineering activities. In the same time, FAMIX was extended to support emerging research interest such as dynamic analysis, evolution analysis, identifier analysis, or bug tracking analysis.

Currently, several research groups from different universities are using Moose as a platform, or FAMIX as a meta-model, and other groups announced interest in using them in the future.

Tudor Gîrba and Adrian Kuhn

Bern, June 20, 2007

Table of Contents

Session 1: Tools and Meta-models

Session 2: Analysis and Visualization

Dynamix - a Meta-Model to Support Feature-Centric Analysis

Orla Greevy

Software Composition Group
University of Berne, Switzerland
greevy@iam.unibe.ch

Abstract

Many researchers have identified the potential of exploiting domain knowledge in a reverse engineering context. Features are abstractions that encapsulate knowledge of a problem domain and denote units of system behavior. As such, they represent a valuable resource for reverse engineering a system. The main body of feature-related reverse engineering research is concerned with feature identification*, a technique to map features to source code. To fully exploit features in reverse engineering, however, we need to extend the focus beyond feature identification and exploit features as primary units of analysis.*

To incorporate features into reverse engineering analyses, we need to explicitly model features, their relationships to source artefacts, and their relationships to each other. To address this we propose Dynamix, am meta–model that expresses feature entities in the context of a structural meta-model of source code entities. Our meta-model supports feature-centric reverse engineering techniques that establish traceability between the problem and solution domains throughout the life-cycle of a system.

Keywords: meta-model, feature analysis, program comprehension, software maintenance

1 Introduction

From an external perspective, users understand a system as a collection of features that correspond to system behaviors to fulfill requirements. As such, features are well-understood abstractions that encapsulate domain knowledge and denote a system's behavioral units. However, the software engineer cannot identify and manipulate features, as they are not explicitly represented in the source code of object-oriented systems. Typically, feature implementation cross-cuts the structural boundaries (*i.e.,* packages and classes) of an object-oriented system [10].

The task of locating the parts of the code that are relevant to a feature in object-oriented systems is widely recognized as a non-trivial task and a body of reverse engineering research collectively referred to as *feature identification* has emerged [1, 5]. A software engineer frequently needs to understand which parts of a system implement a feature to carry out maintenance activities, as change requests and bug reports are usually expressed in terms of features [8]. The main focus of *feature identification* research to date is in a reverse engineering context. Despite the fact that research has highlighted the usefulness of *feature identification* techniques for program comprehension, very little of this effort has found its way into the software engineer's development environment.

Typically the software developer is familiar with a structural representation of a system's source code, for example a UML class diagram. UML also provides sequence diagrams to represent runtime behaviors. To support comprehension, we need to capture the relationship between the structural perspective of a system in terms of software artefacts and the dynamic behavioral entities of an object-oriented system, namely object instantiations and message sends. To represent features we understand dynamic behavior in terms of units that correspond to the features of a system.

In this paper, we describe Dynamix, our meta-model that supports feature-centric analysis of object-oriented systems by focusing on features as first-class entities of analysis in the context of the structural entities, namely packages, classes and methods.

Structure of the Paper. In the next section, we identify the motivation for descibing a meta-model for features. In Section 3 we introduce Dynamix our meta-model for expressing features as first-class entities in the context of a structural model of the source code. We present related work in Section 4 and finally in Section 5 we outline our conclusions.

2 Motivation

Our work is centered around the notion of a feature; we adopt a generally accepted definition as a unit of behavior of a system triggered by the user [5].

The main motivation of our work is to exploit domain knowledge of object-oriented systems inherent in a user's perspective of how a system behaves at runtime so that (1) existing reverse engineering analyses can be enriched with semantic context, and (2) we can define reverse engineering analysis techniques that exploit the notion of features as first-class entities [7]. We establish the goals of a meta-model to express features in the context of a system's behavioral and structural entities as follows:

1. *Behavior.* Due to language features like polymorphism and late binding of object-oriented systems, behavior of a system cannot be completely and automatically determined by analyzing its source code alone. Thus, to capture a system's behavior, we need to perform dynamic analysis and incorporate it in a model of the system.

2. *Combining Static and Dynamic Analysis.* Two main distinct approaches to system comprehension have dominated reverse-engineering research efforts [2]: dynamic analysis approaches and static analysis approaches. Both perspectives are necessary to support the understanding of object-oriented systems [4].

3. *Exploiting Domain Knowledge.* We consider features to be units of behavior encapsulating domain knowledge.

4. *Features as First-Class Entities.* Reverse engineering analysis needs to support system comprehension by breaking the behavior into groupings that reflect its features. To support feature-centric analysis we need to define a meta-model that treats features as first class entities (*i.e.,* primary units) and establishes relationships between features and source artefacts implementing their functionality. Therefore, our model needs to unify behavioral data of features and structural data of source code, thus providing a framework for feature-centric analysis. The model needs to be generic, extensible and should easily accommodate metrics from other feature analysis techniques.

5. *Feature Relevancy Measurements.* Feature identification represents the foundation of our work. Thus, a feature-centric analysis approach needs to provide a measurement to quantify the relevance of a software artefact to a feature, or set of features.

6. *Feature Relationships.* Software engineers need to understand relationships between features, as modifications to one feature may inadvertently affect other features. Furthermore, feature relationships reflect constraints and dependencies in a problem domain. Thus, our meta-model should represent relationships and dependencies between features.

3 Dynamix

We introduce Dynamix, our meta-model to specify behavioral entities of feature execution data and their relationships. Dynamix also specifies the relationships between the behavioral entities and the structural entities representing source artefacts. Dynamix is MOF 2.0 compliant [1]. Our OCL specifications comply with OCL 2.0 [2].

To obtain a model of dynamic and static data of a system under study, we first extract a structural model by parsing a system's source code. We extract *feature traces* by exercising a set of features on an instrumented system. We transform the execution data of feature traces into Dynamix entities and establish the relationships between the execution entities and the source entities of the structural model.

In Figure 1 we show the entities of our model in a UML 2.0 diagram [6]. The *Features* package represents the dynamic behavioral data of the feature traces. The *Structure* package models the entities of the source code. We model behavioral data of features using three entities: *Feature*, *Activation* and *Instance*.

Feature. Each feature trace we capture during dynamic analysis of a system is modeled as a *Feature* entity. A *Feature* entity is uniquely identified by a name. The *Feature* entity allows us to collectively manipulate all the *Activations* that correspond to the events of the feature trace which it models. It maintains a list (modeled as an ordered collection) of all of its *Activations* for ease of manipulation. The first *Activation* of the list represents the root of a feature trace. We assign properties to *Feature* entities based on the *Activations* and their relationships to other entities (*e.g.,* number of *Activations*, number of *Instances* created, number of *Methods* referenced, and feature affinity properties). Relationships between features are shown in the model with a *depends* association. The OCL definition for this relationship between features is given in Figure 2.

Activation. An *Activation* in our model represents a method execution. It holds a reference to its sender *Activation*. In this way Dynamix models the tree structure of a execution trace of a feature preserving the sequence of execution of methods. Time is captured and modeled with two attributes,

[1] http://www.omg.org/docs/ptc/03-10-04.pdf
[2] http://www.omg.org/docs/formal/06-05-01.pdf

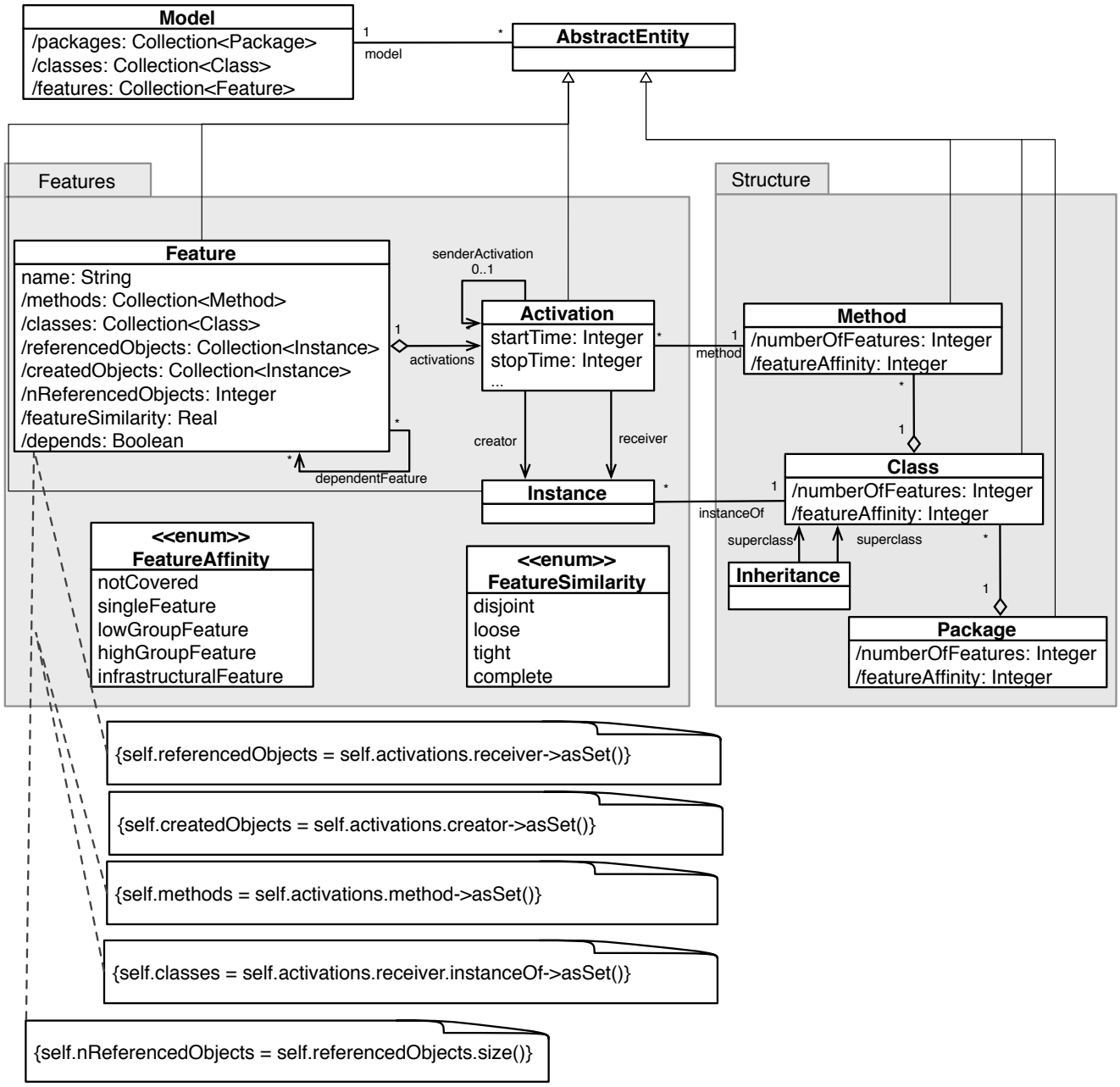


Figure 1. The Dynamix Meta-Model

```
context Feature
  def: importedObjects : Set(Instance) =
         self.referencedObjects->excluding(self.createdObjects)
context Feature
  def: depends(aFeature: Feature) : Boolean =
  ( self.importedObjects->intersection(aFeature.createdObjects)->size() > 0)
               and not (self = aFeature)
```

Figure 2. OCL specification of *depends* relationship between features.

namely *startTime* (*i.e.,* the timestamp in milliseconds, when the method was invoked) and *stopTime* (*i.e.,* the timestamp when it completed execution) . Each *Activation* is associated with a *Method* entity in the structural model. The *Method* entity of the structural model has a relationship to the *Class* entity where it is defined. In this way, we model relationships between features and source entities. Furthermore, an *Activation* is associated with an *Instance* entity which represents the receiver instance of a message. The sender instance is accessible via its sender *Activation*. Thus, Dynamix models the actual object that invokes a method. This does not necessarily correspond to the static relationship between *Method* and *Class* entities, due to inheritance in object-oriented systems. The return value of a message is also stored as a reference to an *Instance* entity in the *Activation* that models the message send.

Instance. We model every instantiated object of a feature trace as an *Instance* entity. An *Instance* is created by an *Activation* and maintains a list of references to all *Activations* that hold a reference to this object (*i.e., Activations* reference the receiver instance of a message, *Activations* that hold a reference to the *Instance* in the return value of a message send). The *Instance* is associated with its defining *Class* entity of the structural model.

AbstractEntity and Model. The entities (Structure and Feature entities) of our model, are derived from *AbstractEntity*. A *Model* comprises every entity, and every entity is associated with the *Model* entity. For example a *Method* entity obtains a collection of all the *Feature* entities in the model via this association.

Dynamix supports feature analysis from different levels of granularity. We exploit relationships between *Feature* entities and source entities to view a system at the package, class or method level of detail.

4 Related Work

Our work builds on feature identification research that deals with locating the implementation of a feature in source code [5,9]. The main variation to our work is in our research focus. We seek extract and model feature entities and establish them as first class entities for analysis of a system from different perspectives.

The work of Deissenboeck and Ratiu emphasizes the role of a unified meta-model to exploit the concepts for reverse engineering [3]. Similarily, we identify the need to model features in the context of a structural model of source code as the basis of our work.

5 Conclusion

To fully exploit features in reverse engineering, we need to treat features as primary units of analysis. We motivated the need to describe a meta-model for features : (1) to enrich reverse engineering analysis techniques that extract structural views of a system with semantic knowledge about roles of source artefacts in features of a system, and (2) to reason about a system in terms of features themselves and relationships between features.

We describe Dynamix, a meta-model that expresses the execution entities of feature behavior and their relationships. Furthermore our meta-model expresses the relationships between the execution entities and a structural model of source code. Dynamix supports analysis that combines static and dynamic views of a system.

References

[1] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366, Los Alamitos CA, Sept. 2005. IEEE Computer Society Press.

[2] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[3] F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06)*, 2006.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press. Also appeared in ACM SIGPLAN Notices 35 (10).

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[6] M. Fowler. *UML Distilled*. Addison Wesley, 2003.

[7] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[8] A. Mehta and G. Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.

[9] N. Wilde and M. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[10] E. Wong, S. Gokhale, and J. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

ScheMoose - Supporting a Functional Language in Moose

Katerina Barone-Adesi *and* Michele Lanza
Faculty of Informatics - University of Lugano, Switzerland

Abstract

The Moose Reengineering environment has traditionally been targeted at object-oriented languages with little to no support for other paradigms, although this is theoretically supported too. In this article we describe the experiences we obtained while implementing ScheMoose, a prototype parser and MSE exporter of Scheme code written itself in Scheme. We describe to what extent ScheMoose could make use of the FAMIX metamodel, in which cases we needed to extend the meta-model and also provide a validation of ScheMoose in the form of Mondrian visualizations of Scheme programs.

1 Introduction

Recently, functional programming languages such as Scheme and LISP are being reconsidered, due to their elegance and the abstraction capabilities that they offer. The simplicity of a functional language like Scheme makes it an ideal tool to teach the fundamentals of programming, as we do ourselves at our university.

The ScheMoose project's motivation lies in the many systems that we are being delivered by the students and by the lack of comprehension tools that are available for Scheme. Our goal is to reuse the existing infrastructure of the Moose reengineering environment [1], by extending it with support for functional languages. This involves the following steps, which also roughly dictate the structure of the paper:

- The creation of a parser for Scheme code written in Scheme and the implementation an exporter in Scheme which exports the modeled code to the mse format
- The subsequent modeling within Moose and the necessary extensions/additions to Moose to better cope with functional source code
- The final validation of ScheMoose by means of visualizations obtained using the Mondrian framework [2].

2 ScheMoose

Schemoose is a Scheme program which extracts information from Scheme source code and exports it as an mse file, using the FAMIX metamodel. It models functions and variable definitions, project structure (i.e., which definitions are in which file), and function calls. It also determines some semantic information about the code, such as whether a function is a predicate function.

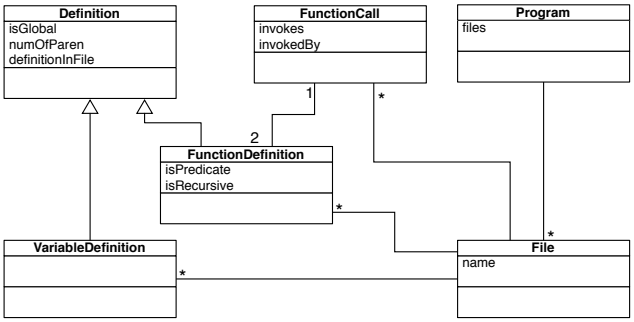


Figure 1. ScheMoose metamodel

We are currently adding semantic information to the extracted model, such as whether a function is directly recursive (it calls itself) or is mutually recursive with another functions, etc.

In Figure 1 we see a reduced class diagram of the information we currently model. We expect to soon finalize the amount of information that we extract from Scheme code. In terms of metrics we currently defined the number of parentheses (NPAR) and the position within the file (DIF). While the first metric is concerned with modeling the complexity of the function, we plan to exploit the DIF metric to obtain a sequential view of the code, since evaluation order in Scheme is a primary concern, and also reveals information about the way the code has been written.

ScheMoose, itself written in Scheme, makes use of the Scheme function `read` to extract the information.

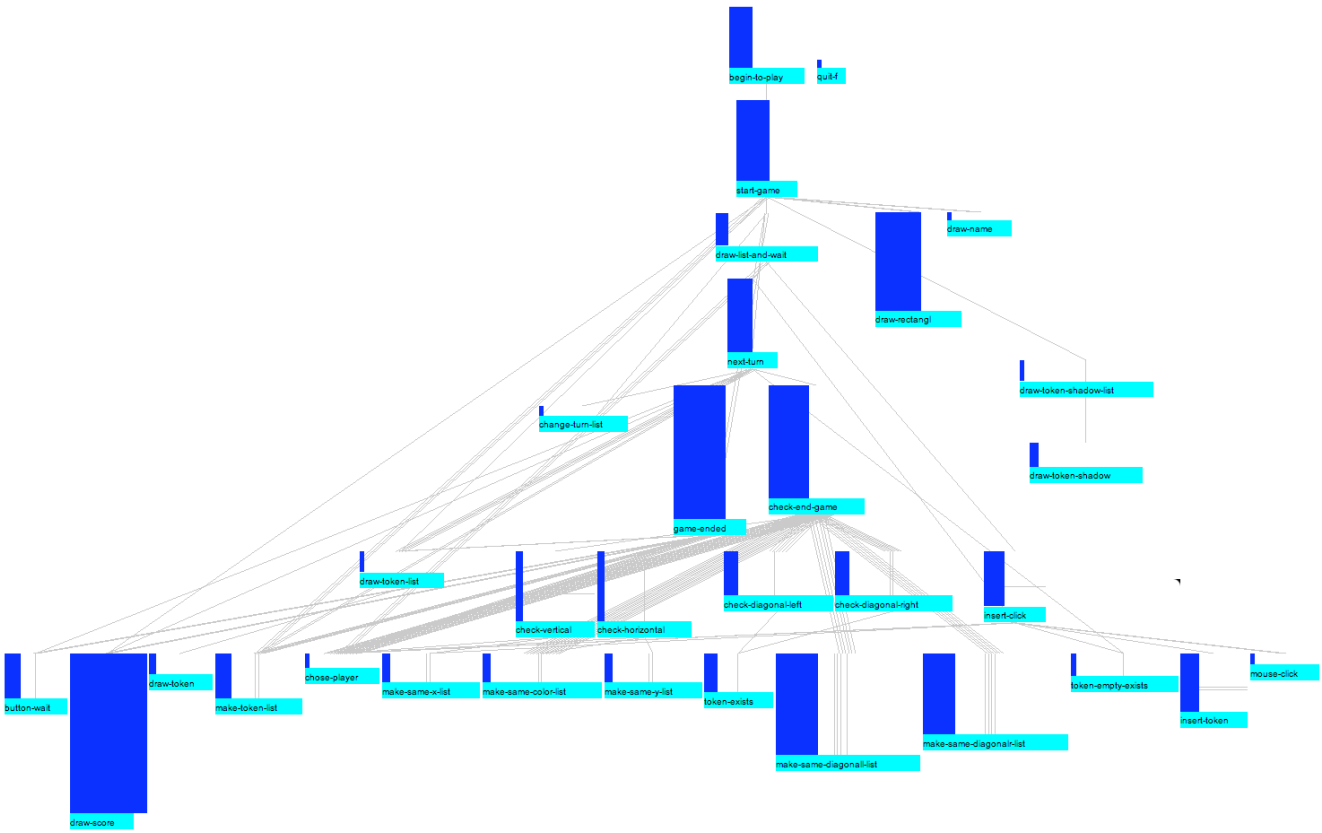


Figure 2. Mondrian visualization of a Connect-4 game written in Scheme

3 Enriching FAMIX

To a large extent Moose if already apt at handling functional languages, i.e., in our specific case we can reuse the existing FAMIXFunction and FAMIXInvocation classes.

ScheMoose is work in progress, and because of that we only extended FAMIXFunction with the meta-description of "isPredicate", "numberOfParentheses", and "isDirectlyRecursive" so far.

4 Applications

We used Mondrian to visualize the code of one of the student projects written during the course "Programming Fundamentals".

In Figure 2 we see the whole program displayed as a tree of functions that invoke each other. The boxes display information about each function, namely the number of outgoing invocations for the width, and the number of parentheses for the height.

5 Conclusions

Our preliminary experiences, since the ScheMoose project is still going on, is that Moose is fit for modeling functional languages, and that the necessary extensions to some of the FAMIX classes can be considered marginal.

There are still a number of things we plan to implement, such as supporting macros, higher-order functions, continuations, conditional expressions, etc.

References

[1] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[2] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, pages 135–144, New York, NY, USA, 2006. ACM Press. To appear.

Making FAMIX Test-Aware

Bart Van Rompaey
Lab On Re-Engineering
University Of Antwerp
bart.vanrompaey2@ua.ac.be

Abstract

The gaining popularity of agile development has resulted in test suites encompassing substantial parts of the overall system size. As unit tests are typically specified in the same programming language, a re-engineering meta-models such as FAMIX *works on these test suites as well. Yet such model entities do not bear information about the semantic value in a software testing context. In this paper we propose* FAMIX-XUNIT *as an extension that enables the analysis of test suites and the interrelationship with the production system.*

1 Introduction

Development methodologies such as eXtreme Programming or test-driven development result in lots of small unit tests that are introduced early in the life cycle and further on are frequently modified, co-evolving alongside production code. As such, test suites degrade over time as well, thereby requiring similar re-engineering efforts.

When capturing a model from a system's source code, the test suite can be captured along for further analysis. However, as software tests require other design characteristics than a production system, "standard" problem detection and resolution strategies do not suffice to check and improve a test suite's quality. By making the re-engineering meta-model test-aware, we can perform system analyses that involve the test suite as well as the interaction between the test suite and the production system, such as:

- Recovering the design of a test suite by identifying test cases, units under test, abstractions in the tests, reusable test facilities, etc.
- Checking for signs of design erosion hindering test understanding and evolution, for error-prone constructs or for duplicate tests [3, 5].
- Navigating between production entities and test entities to study the intent of a particular test or, inversely, to identify test cases covering a particular unit.

To separate software tests analysis from regular code, in this paper we propose FAMIX-XUNIT as a simple yet powerful FAMIX (see [2]) extension that bears information about test suites and the concepts within. The extension is based upon common implementations of the XUNIT family of testing frameworks [4], the *de facto* standard for (originally) unit testing, but expanded upon to accommodate other testing techniques as well.

The operations that are required from a model to support these analyses are comparable to those described for a regular FAMIX model.

2 Test Concepts

The central notion in software testing is the Setup-Stimulate-Verify-Teardown (S-S-V-T) cycle [1], describing the execution of a single test. During the first step of a test case's execution, the setup, the necessary resources to perform a test are acquired and instantiated. Next, a stimulus (a message) is sent to the unit under test by the test case. This stimulus is expected to provoke a certain reaction. This result is queried for by the test case and compared with the expected result. Finally, all acquired resources are released again during tear down.

In previous work we formalized the test concepts that occur in this S-S-V-T cycle [5]. We informally describe them here and illustrate using JUNIT[1]:

- a *Test Case* groups a set of tests performed on the same unit under test. In JUNIT, test cases extend `junit.framework.TestCase`.
- a *Test Command* is a container for a single test. It is typically implemented as a parameterless `public void` method of a test case with a name prefix `test`. This method contains the stimulation and verification phases of the S-S-V-T cycle.
- *Setup* and *Teardown* are methods of a test case that are executed before and after every test command. The JUNIT implementation is a parameterless `public void` method named `setUp` and `tearDown` respectively.

[1] an XUNIT implementation for Java. from JUnit 4 on, annotations can be used as an alternative to naming conventions

- a *Test Case Fixture* is the set of attributes a test case requires to bring the unit under test into a desired initial state. It can contain both instances of the units under test as well as some shared data objects.
- a *Stimulus* is either a method invocation or an attribute access, from a test case to production code.
- an *Assert* (also called *Check*) is a invocation of a test framework method that compares the actual result with the expected one. JUNIT has a set of such methods in `junit.framework.TestCase`, providing multiple comparison mechanisms.
- a *Test Helper Method* is a method belonging to a test case that is assisting test commands in recurring (possibly parametrized) stimulation or verification.
- a *Test Suite* is a container for a set of test cases that are executed together. The JUNIT implementation is a `public static` parameterless method named `suite`, returning a `junit.framework.Test` object.

3 Extending FAMIX

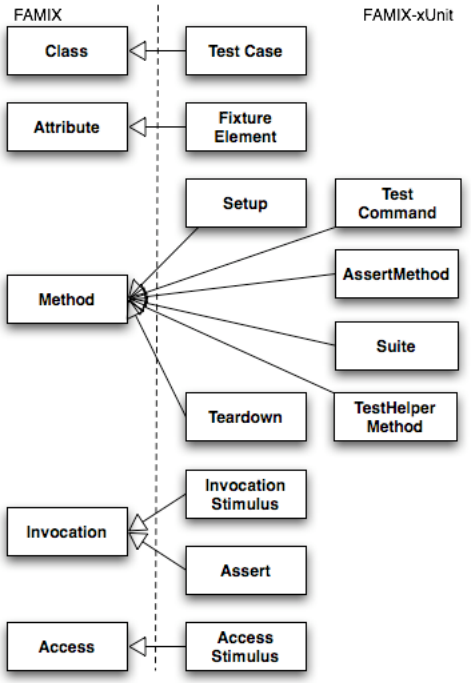
Making FAMIX test-aware requires no more than making the semantic value (i.e. the role an entity fulfills during software testing) of test entities explicit. As a first extension option, model entities can be given additional, optional properties that explain their role during testing, e.g. a CLASS entity has an optional field ISTESTCASE indicating whether a particular class represents the role of a test case in the test suite. We opted, however, for adding new entities: by introducing model entities as subclasses to determine testing roles, we (i) avoid cluttering existing entities with additional optional properties and (ii) end up with a solution that allows elegant model queries and enumerations over sets of test entities. This results in the model extension based on inheritance as shown in Figure 1. The presented mapping between testing concepts and object-oriented model classes holds for most common xUnit implementations such as JUNIT for Java, CPPUNIT for C++ and NUNIT for C#.

FETCH[2], the Fact Extraction Tool Chain developed at the University of Antwerp, supports FAMIX-XUNIT for test suite analysis. The refinement of a regular FAMIX model to a test-aware model is implemented as a post-processing step after capturing a model from source code. The characteristics described in Section 2, already represented in a regular FAMIX model, serve as heuristics to identify test entities.

4 Conclusion

In this paper we first discussed the major domain-specific concepts in software testing. Next, we proposed an extension to FAMIX, FAMIX-XUNIT, that allows us to model test suites, by means of these concepts modeled as subclasses of regular FAMIX classes. This approach results in a simple yet powerful backwards compatible model that is moreover easy to obtain, paving the way for research in test suite re-engineering and co-evolution analysis.

Figure 1. FAMIX-xUnit



Acknowledgments – This work was executed in the context of ITEA project if04032 entitled *Software Evolution, Refactoring, Improvement of Operational&Usable Systems* (SERIOUS) and has been sponsored by IWT, Flanders.

References

[1] K. Beck. Simple smalltalk testing: With patterns. *The Smalltalk report*, 4(2):16–18, 1994.

[2] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0: The FAMOOS information exchange model. Technical report, Universität Bern, 1999.

[3] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok. Refactoring test code. In *Proc. Extreme Programming and Flexible Processes*, pages 92–95, 2001.

[4] P. Hamill. *Unit Test Frameworks*, chapter 3: The xUnit Family of Unit Test Frameworks. O'Reilly, 2004.

[5] B. Van Rompaey, B. Du Bois, and S. Demeyer. Characterizing the relative significance of a test smell. In *Proceedings of the 20th Int'l Conference on Software Maintenance*, 2006.

[2] http://www.lore.ua.ac.be/Research/Artefacts/fetch/

Teaching FAMIX about the Preprocessor

Matthias Rieger, Bart Van Rompaey
Lab On Re-Engineering
University of Antwerp, Belgium
{matthias.rieger,bart.vanrompaey2}@ua.ac.be

Roel Wuyts
IMEC vzw.
Leuven, Belgium
wuytsr@imec.be

***Abstract*—The C Preprocessor (cpp) is a macro processor which is used extensively for C and C++ programs to store, for example, multiple configurations of a system in the same source files. It is also known that undisciplined use of the preprocessor can lead to difficult to understand code. Hence there is a need for program analysis emphasizing preprocessor usage, which in turn requires explicit preprocessor information. This paper proposes to extend the FAMIX metamodel with a small number of entities that enable analysis of all configurations of a system, as well as of the use of the macro feature of cpp. We thereby can improve regular analysis as it is done today, and also enable new types of analysis not currently possible.**

I. INTRODUCTION

The cpp is a simple but powerful tool which is has been in a symbiotic relation with the C compiler from the beginning. Many shortcomings of the C language can be resolved with preprocessor directives. The fact that it does not respect the C syntax rules and its great flexibility make cpp however a potential source of many suboptimal coding practices and the ensuing confusion. cpp is a necessary evil for every C programmer and maintainer. Even the advent of C++, which solves many shortcomings of C that were handled with preprocessor directives, did not eradicate the use of cpp from the programmers toolbox.

From the program understanding viewpoint, cpp complicates the analysis of source code considerably. The programmer actually writes two programs: the cpp program and the C program. The result of the cpp program can be one of a multitude of C programs. Someone wanting to parse a system for program understanding purposes has two options: either choose one particular configuration and parse the preprocessed code for that configuration, or make the parser more robust so that it understand the unpreprocessed code, which does not have to be valid C.

Up until recently, Information on the preprocessor directives has typically not been made part of the metamodels used in program understanding. This is understandable as the preprocessor directives form a separate language which cannot be expressed with the model entities of a language like C or C++. The problems with neglecting this kind of information are however apparent:

- Having only a partial view upon the system, if the parser only sees the code of a single configuration.
- Having multiple definitions of the same entity if the parser gets to see code from all configurations.
- Missing invocations and accesses that are only assembled during macro expansions.

Not having any information about the preprocessor directives also prevents analysis of the cpp program, which can be of considerable complexity and in need for refactoring itself.

In this paper we propose to extend the FAMIX metamodel[1] with the information necessary to represent a complete system, including all configurations, and enough information about the cpp program so that it can be analyzed as well.

The structure of the paper is as follows: Section II presents other approaches to include preprocessor information in regular program analysis. Section III lists a number of analysis questions that we want to be able to answer with the new information. The main Section IV lists the preprocessor directives we want to model and details the extensions to the FAMIX model. In Section V we give an initial ideas on how to make parsers capable of extracting the combined information.

II. RELATED WORK

Whereas the standalone analysis of the cpp has already a longer history ([2],[3],[4]), integrating information from the preprocessor and the parser has only recently been taken up.

Garrido[5] describes a parser and an AST which incorporates both preprocessor directives and normal C code. This enables refactorings on the complete system over all configurations and including macros.

L.Vidács and Á.Beszédes[6] describe a schema for storing preprocessor information. Their model represents all preprocessor directives in detail, enabling to extract the original source code from the model again. The preprocessor schema is extending the COLUMBUS schema[7].

Similar to the COLUMBUS schema, Doum et al. [8] are integrating preprocessor information into the *Intermediate Language*[9] of Project Bauhaus.

III. ANALYSES ON THE WHOLE SYSTEM

To motivate our work we now list potential analyses that we want to enable once a model is enriched with information about the preprocessor directives. We generally aim to extend regular FAMIX queries onto the whole system, and we are interested in the interplay between cpp and C code.

- Which invocations are only executed when the DEBUG or the TEST flag is set?
- In which configurations is this variable accessed?
- How many times is this macro instantiated, and where?

- Are there invocations which are either to a function, or are a macro expansion, depending on the configuration?
- Which macros are used in `#if` directives but never defined in the source, i.e. which are the free variables that are set at compile time?
- Which macros are defined and used in mutual exclusion? Eventually one could want to build a lattice of all potential configurations in the system, as was proposed by Krone&Snelting[2].

There are also a number of tasks that we do not aim to be able to perform with our model extensions, like for example recreating the original source code from the model. We will therefore neglect some of the information available from preprocessor directives.

IV. Modeling Preprocessor Information

This section introduces the new FAMIX entities that model information from a selection of all existing preprocessor directives. Our goal is to extend FAMIX only with a minimal number of new entities and attributes. We therefore do not add representations for each directive, and we also do not represent all information that can be found in the preprocessor directives.

We basically focus on modeling *Conditional Compilation* (Section IV-A) and the use of *Macros* (Section IV-B). Section IV-C lists, for completeness, the directives which we ignore.

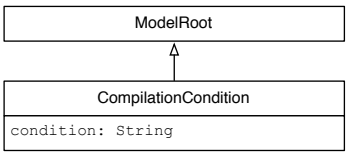
A. Modeling Conditional Compilation

Conditional directives are used to provide different parts of the source code to the compiler, depending on the setting of some configuration variables. To model conditional compilation in FAMIX we extract information from the occurrences of the following directives:

`#if,#ifdef,#ifndef,#elif,#else,#endif`

We do not model these directives directly, however. For our analysis needs, we propose to add one new entity and one attribute to an existing entity.

Introducing New FAMIX *Entities:* We introduce the entity COMPILATIONCONDITION.



The attribute `condition` contains a conjunction of all conditions that are in effect at the given point in the source code. For entities which are not constrained by condition, i.e. which are part of every configuration, `condition` is set to `true`. As an example, consider the following code:

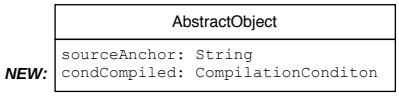
```
#ifdef FILECHECK
#ifdef VMS
  if (strchr (name, ':') != 0)
#else
  if (absoluteName())
#endif
#endif
```

This code is modeled with an ACCESS to the `name` variable, which is tagged by a COMPILATIONCONDITION containing "FILECHECK && VMS", and an INVOCATION of the function `absoluteName()`, tagged with "FILECHECK && ! VMS". Note how nested conditions are conjuncted, and how the condition of the `#ifdef` branch gets negated in the `#else` branch.

We have chosen to store a condition as its own entity, instead of simply storing the condition string in every FAMIX entity. The reason is that by letting entities refer to explicit and shared COMPILATIONCONDITIONS, analysis of the conditional structure of the system can proceed faster.

Note that COMPILATIONCONDITION is made a subclass of MODELROOT, which implies that there is no `sourceAnchor` associated with it. Information about where (in terms of source ranges) a condition holds is therefore not explicitly retained in the model. We do not consider this to be a drawback, as FAMIX in general tends to abstract from concrete source file locations.

Modifying Existing FAMIX *Entities:* The entity ABSTRACTOBJECT gets a new attribute `condCompiled`, which points to a COMPILATIONCONDITION.



Adding the condition at such a high level in the model means that all FAMIX entities are tagged with such a condition. This makes sense, as the FAMIX entities are all extracted from the output of the preprocessor, i.e. they are all potentially subject to conditional compilation.

Sidenote: Adapting the FAMIX *Unique Name:* FAMIX uses a unique naming scheme—the intrinsic name of each program name—to ensure that entities created by different tools have the same name. Extracting the elements from all configurations of a system with conditional compilation may result in name clashes as we can see in the following code example:

```
#ifdef HIGH_END
  long  count;
#else
  int   count;
#endif
```

Here, the model will contain two entities with the same name in the same scope. To distinguish them we therefore propose to extend the unique name with a postfix that shows the condition under which the entity is part of the running system. If the variable `count` is part of the method `foo()`, we would get the two following unique names:

```
MyClass::foo().count:HIGH_END
MyClass::foo().count:! HIGH_END
```

In the case of the the default condition—`true`—we can ignore the prefix and just write the unique name as before.

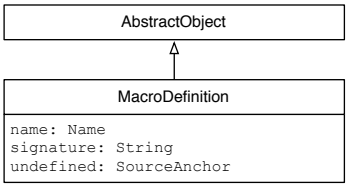
It seems advantageous to append the condition to the name, as conditions can be of varying length and the most interesting part of the unique name is still the name of the program entity. Since conditions can become very long and complicated, an alias scheme might be used so that a human reader can more easily compare the names.

B. Modeling Macro Usage

Extracting the preprocessor information that deals with macros entails handling macro *definitions* and macro *expansion*.

Modeling Macro Definitions: `cpp` understands the directives `#define` and `#undef` for the definition of macros. Macros are commonly distinguished as *object-like*, having nor parameters, or being *function-like*, which means that they have a number of parameters. We represent both types of macros with the same FAMIX entity, similar to regular functions where the number of parameters does not prescribe the entity type either.

Introducing New FAMIX *Entities:* To represent macro definitions, we propose to introduce the entity MACRODEFINITION.



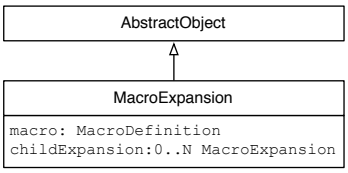
The attribute `name` is the name of the macro. The `signature` gives a list of the parameters. If the macro is *object-like*, the `signature` is empty, e.g. "()". The location of the definition of a macro is recorded in the `sourceAnchor` that is inherited from ABSTRACTOBJECT. The `undefined` attribute records if the macro is undefined at some later location. Note that since a macro can be redefined after an `#undef` directive, it is possible to have macros with the same name but differing implementations valid in separate parts of the source code. The model will contain a MACRODEFINITION for each range in which the macro is valid.

Note that we do not store the source code of the macro, nor do we perform any extraction of nested macro invocations in the macros' source. The reason is that at the point of definition of a macro, we do not know which tokens will defined to be a macro at the point of expansion.

Modeling Macro Expansions: Macro expansions are not indicated by a specific directive[1]. An expansion is triggered when a token is encountered for which an entry exists in the symboltable of the preprocessor.

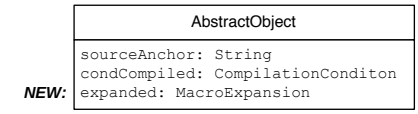
Our goal is to extract not only the fact of the macro expansion—similar to extracting a function invocation—but also to extract all the normal FAMIX entities from the expanded code. Because of the string concatenation operator `##`, which may create identifiers that are not visible in the source code, we actually have to perform a full expansion of every macro encountered if we want to get the complete set of program entities.

Introducing New FAMIX *Entities:* We introduce the MACROEXPANSION which represents an expansion of a single macro:



The attribute `macro` points to the definition of the macro that we're expanding. Since an expansion can entail other macros being expanded, the attribute `childExpansion` contains all expansions triggered by running the preprocessor over the expanded macro code. This results in a tree of macro expansions.

Modifying Existing FAMIX *Entities:* To ensure the link between the program code and the preprocessor code, all FAMIX entities extracted from code that is the result of a macro expansion must be tagged as such. Since every type of FAMIX entity can result from a macro expansion, we add the flag high in the FAMIX class hierarchy.



The `expanded` attribute points to the root of the expansion tree. The default value of the `expanded` for entities that are extracted from unexpanded code is `null`.

Some FAMIX entities describe entities which may be only partially be the result of a macro expansion. Consider for example the following code fragment:

```
#define A int
 A c;
```

We extract a LOCALVARIABLE with `declaredType=int` and `name=c`. Only the `type` is the result of an expansion, but we mark the entire entity as being `expanded`. Finer grained analysis is left to tools which want to investigate the exact nature of the expansion. We estimate the simplification of the model higher than the sacrifice of some accuracy.

Note that the MACRODEFINITION entity, as a subclass of ABSTRACTOBJECT, inherits the `expanded` attribute as well. Macro definitions cannot, however, be the result of a macro expansion. It is debatable if this inconsistency should lead us to introduce a separate subclass of MODELROOT to be the root of preprocessor entities such as MACRODEFINITION and MACROEXPANSION, which after all, are fundamentally different entities than CLASSES and METHODS.

[1]The *concatenation* `##` and the *stringification* `#` directives are exceptions to this rule. They can, however, only occur in macro code.

C. Preprocessor Directives Not Modeled

To keep the extensions to the FAMIX metamodel minimal, we exclude certain preprocessor directives from being represented in a model. We are guided by the types of analysis described in Section III when choosing to exclude the following directives that do not contribute to the analyses:

`#include` : The include relationship between files is already represented by the INCLUDE entity[10].

`#pragma`,`#sccs`,`#ident` : These directives allow the programmer to convey platform specific information to the compiler and some of them are non-standard. We currently deem this type of information too low level and choose not to model it.

`#error` : This directive stops the preprocessor and reports a fatal error, which is potentially interesting information for program analysis. Not modeling it is due mainly to reasons of space. It would however be straightforward to add a model entity for this simple directive.

`#line` : This directive allows the compiler to keep track of the current line-number. We do not model it as its information is already being incorporated into the FAMIX source anchors.

V. EXTRACTING THE INFORMATION

This section shows that is possible, in principle, to extract the information that we want to model from the source code. Parsing the `cpp` and C/C++ code at the same time has been worked on before. Until recently, however, the proposed solutions were either incomplete (only dealt with conditional compilation, for example) or based on heuristics.

G.Badros&D.Notkin[11] have built an integrated preprocessor/parser framework which can be customized to analyze C code and preprocessor directives at the same time. The symbol table of the preprocessor is accessible and the user is able to feed code back to the preprocessor and parser. It seems therefore possible to parse all configurations using this setup.

A.Garrido[5] aims at being able to perform refactoring in the presence of preprocessor directives and over the whole system. She describes the semantics of a parser which understands and executes preprocessor directives as well as regular C code. She is able to construct an AST which represents all possible configurations of the systems at once.

M.Latendresse[12] has described three related rewrite systems which symbolically execute the `cpp` program. His system is able to find for every line of code the precise conditions under which the line is reached. It is also able to determine for each line all possible values for the preprocessor variables. This will allow to perform the necessary expansions to recreate all configurations.

It therefore seems like the parsing problem is solvable, and we are able to populate a FAMIX model with the information about an entire system.

VI. CONCLUSIONS

C and C++ systems rely on the preprocessor to roll multiple configurations of a system into one. Analysis of preprocessor usage is therefore needed to enable understanding of the semantics of the system, something which was lacking from the FAMIX model.

We have proposed a FAMIX extension, consisting of three new entities, and three new attributes for MODELROOT and ABSTRACTOBJECT. With these extensions, analysis that takes preprocessor directives into account becomes possible. As an example, the usage of macros can be investigated in a similar way as its currently done for functions. The relationship between macros and the code they generate is opened for analysis, as well as the structure of the configuration space.

The extensions we have proposed are backwards compatible: if the parser is fed with preprocessed code, the extracted model is the same as with the unextended model, except for some attributes holding default values.

We recognize that in order to make these model extensions operational, considerable effort is required to build up the parser infrastructure.

ACKNOWLEDGMENT

This work was executed in the context of the ITEA project if04032 entitled *Software Evolution, Refactoring, Improvement of Operational&Usable Systems* (SERIOUS) and has been sponsored by IWT, Flanders.

REFERENCES

[1] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 — The FAMOOS Information Exchange Model," University of Bern, Tech. Rep., 2001.

[2] M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 49–57.

[3] P. E. Livadas and D. T. Small, "Understanding code containing preprocessor constructs," in *Proceedings IEEE Third Workshop on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 89–97.

[4] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *IEEE Trans. Softw. Eng.*, vol. 28, no. 12, pp. 1146–1170, 2002.

[5] A. Garrido, "Program refactoring in the presence of preprocessor directives," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Aug. 2005.

[6] L. Vidács, A. Beszédes, and R. Ference, "Columbus schema for C/C++ preprocessing," in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.

[7] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus - reverse engineering tool and schema for C++," in *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. Washington, DC, USA: IEEE Computer Society, Oct. 2002, pp. 172–181.

[8] M. B. Doum, H. Holle, P. Weder, and H. Wiemann, "Projekt ArteFakt, Semesterbericht AG Präprozessor," Oct. 2006, Universität Bremen, Germany.

[9] J. Czeranski, T. Eisenbarth, H. M. Kienle, R. Koschke, E. Plödereder, D. Simon, Y. Zhang, J.-F. Girard, and M. Würthner, "Data exchange in Bauhaus," in *Proceedings WCRE '00*. IEEE Computer Society Press, Nov. 2000.

[10] H. Bär, "FAMIX C++ language plug-in 1.0," University of Bern, Tech. Rep., Sep. 1999.

[11] G. J. Badros and D. Notkin, "A framework for preprocessor-aware C source code analyses," *Softw. Pract. Exper.*, vol. 30, no. 8, pp. 907–924, 2000.

[12] M. Latendresse, "Rewrite systems for symbolic evaluation of C-like preprocessing," *csmr*, vol. 00, p. 165, 2004.

The Metabase: Generating Object Persistency Using Meta Descriptions

Marco D'Ambros *and* Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Martin Pinzger
s.e.a.l. - software evolution and architecture lab
University of Zurich, Switzerland

Abstract

Because of its language independent metamodel FAMIX, the Moose environment allows researchers to analyze software systems in a uniform way and to exchange data and results. To support interoperability, different interchange file formats such as XMI, CDIF, and recently MSE, have been proposed and used. The use of text files, to exchange data, has three major issues: (1) the files containing the model of a system should always be parsed before being used in Moose, (2) the files must be parsed entirely, even if only a small part is needed, and (3) compatibility is a concern since the old interchange formats are no longer supported in recent versions of Moose.

We propose an alternative approach for FAMIX data exchange, based on object persistency. FAMIX models are stored in a database in a transparent way and they can be remotely accessed. Model entities are retrieved from the database "on-demand", i.e., only the parts needed are read and not the entire model. The mapping between Smalltalk objects and database rows is automatically generated from a meta description, which does not limit the approach to the FAMIX meta model, but to any *meta model defined by a meta description.*

1 Introduction

The FAMIX meta model [1] has been used for several years by researchers to exchange descriptions of software systems. Using a language independent meta model allows them to analyze software systems written in different languages, to apply several tools, and to compare results. In the context of the Moose framework, different file interchange formats have been used to exchange FAMIX models. The first two, *i.e.,* CDIF and XMI [3], are not supported any more in recent versions of MOOSE. The last, and currently used, file format is MSE[1].

We identified three major issues in using text files for exchanging models:

1. *Parsing*. The text file containing the model has to be parsed to import the model into the Moose environment.

2. *Performance*. It is not possible to import only parts of the model, *i.e.,* parse only parts of the text file: The entire text file has to be parsed. For models of large software systems this can have a severe impact on performance.

3. *Compatibility*. Recent versions of Moose do not support old interchange file formats anymore. This has a strong impact on tools based on old versions of Moose.

We propose an approach to exchange models, based on object persistency instead of text files. The technique reads/writes objects from/to a database in a transparent way, avoiding import/export operations. The objects are retrieved "on demand," without the need of reading the entire model.

The persistency mechanism, *i.e.,* the mapping between the database and the actual objects, is automatically generated from a meta model description written in Meta.[2] Therefore, the approach is not limited to FAMIX, but can be used for any meta model described by Meta.

The approach is implemented in Smalltalk, in the context of Moose, using GLORP [2] for the object persistency. The concepts presented in this paper are strictly related to this language and environments, but they can be generalized to other languages and environments, *e.g.,* Java and Hibernate.[3]

Structure of the Paper. In Section 2 we introduce the object persistency technique based on GLORP. We then present our Metabase approach in Section 3 and we provide an example in Section 4. We conclude in Section 5 by summarizing our contributions and discussing future work.

[1] See http://smallwiki.unibe.ch/moose/mseformat/

[2] http://smallwiki.unibe.ch/moose/tools/meta/

[3] http://www.hibernate.org/

2 Object Persistency

The Metabase relies on GLORP[4] for object persistency. GLORP is a simple but powerful object-relational mapping layer for Smalltalk. It allows us to define the mapping between Smalltalk objects and table and rows in a relational database (DB). Once this mapping is defined, objects can be read from and written to the DB in a completely transparent way, without having to write any SQL statement.

Example. We want to define the mapping for simplified versions of FAMIXClass and FAMIXMethod. FAMIXClass has a name, belongs to a package and has a collection of methods, while FAMIXMethod just has a name. We need to create a new class, *i.e.,*`FamixDescriptorSystem`, inheriting from `Glorp.DescriptorSystem`. In this class we add methods to define the structure of the database table corresponding to the FAMIX class and method and to define the mapping between the tables and the classes. This is shown in the following code snippets.

```
tableForFAMIXClass: aTable
 aTable createFieldNamed: 'Id' type: platform serial.
 aTable createFieldNamed: 'Name'
   type: (platform varChar: 50).
 aTable createFieldNamed: 'PackagedIn'
   type: (platform integer).

tableForFAMIXMethod: aTable
 aTable createFieldNamed: 'Id' type: platform serial.
 aTable createFieldNamed: 'Name'
   type: (platform varChar: 50).
 aTable createFieldNamed: 'BelongsTo'
   type: (platform integer).

descriptorForFAMIXClass: aDescriptor
 | t |
 t := self tableNamed: 'Class'.
 tMethod := self tableNamed: 'Method'.
 tAttribute := self tableNamed: 'Attribute'.
 tPackage := self tableNamed: 'Package'.
 aDescriptor table: t.
 aDescriptor addMapping:
   (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
 aDescriptor addMapping:
   (DirectMapping from: #name to: (t fieldNamed: 'Name')).
 (aDescriptor newMapping: OneToOneMapping)
   attributeName: #packagedIn;
   referenceClass: FAMIXPackage;
   mappingCriteria: (Join from: (t fieldNamed: 'PackagedIn')
     to: (tPackage fieldNamed: 'Id')).
 (aDescriptor newMapping: OneToManyMapping)
   attributeName: #methods;
   referenceClass: FAMIXMethod;
   join: (Join from: (t fieldNamed: 'Id')
     to: (tMethod fieldNamed: 'BelongsTo')).
 ^aDescriptor

descriptorForFAMIXMethod: aDescriptor
 | t |
 t := self tableNamed: 'Method'.
 aDescriptor table: t.
 aDescriptor addMapping:
   (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
 aDescriptor addMapping:
   (DirectMapping from: #name to: (t fieldNamed: 'Name')).
```

In the code snippet we see three kinds of mapping: Direct, one-to-one and one-to-many.

Direct Mapping. It is used to express simple relationships between instance variables and table columns. It is used when the "type"[5] of the instance variable is directly supported by the DB, for example for Integer, Text, Varchar, Timestamp, *etc.*

One-to-one Mapping. It is used to express the relationship between FAMIXClass and FAMIXPackage. This mapping, declared in the FAMIXClass descriptor, defines the following properties:

- The attribute name: The name of the instance variable getter.
- The reference class: Specifies the class of the objects, and therefore the corresponding table in the DB. In the considered case the class is FAMIXPackage, which corresponds to the Package table (not shown for brevity).
- The join expression: Defines which columns of the two tables are linked. This information is used by GLORP to create the appropriate SQL join query to fetch the data from the DB and create the objects.

One-to-many Mapping. It expresses that a FAMIXClass can have several FAMIXMethods. The structure of the mapping is the same as the one-to-one mapping, with two differences. First, the attribute name refers to a collection of objects instead of a single object. All these objects have to be instances of the class "referenceClass" (FAMIXMethod). Second, the data will be written in the table corresponding to the reference class (FAMIXMethod), instead of the current class (FAMIXClass). This is because each row in the method table refers to a row in the class table (the container class), while each row in the class table can refer to multiple rows in the method table.

A last type of mapping, not used in the code snippet, is *many-to-many*. It expresses the most generic relationship by means of a link table. If two classes have this kind of relationship, the relationship itself is stored in a separated link table in the DB.

What about Inheritance? We want to add to our simplified FAMIX model a superclass of FAMIXClass, namely FAMIXAbstractNamedEntity, which has a name as instance variable. When adding this superclass, we also remove the *name* instance variable from the FAMIXClass class, since it is inherited from FAMIXAbstractNamedEntity. GLORP

[4]Generic Lightweight Object-Relational Persistence. For details refer to [2] and to the GLORP web site http://www.glorp.org/

[5]To use GLORP we have to assume that an instance variable is always of the same class, called type.

provides two techniques to manage inheritance: Filtered and Horizontal. In the filtered inheritance all of the classes are represented in a single table, with a discriminator field for which subclass they are. The table has the union of all possible fields for all classes. In the horizontal inheritance each concrete class is represented in its own table. Each table will duplicate the fields that are in common between the concrete classes. Figure 1 shows the two approaches for our examples.

AbstractNamedEntity	
Id	Name
1	EntityA

Class		
Id	Name	PackagedIn
1	ClassA	PackageA

(a) Horizontal Inheritance

AbstractNamedEntityAndClass			
Id	Name	PackagedIn	Class
1	EntityA	-	FAMIXAbtractNamedEntity
2	ClassA	PackageA	FAMIXClass

(b) Filtered Inheritance

Figure 1. Types of inheritance in GLORP

In horizontal inheritance (Figure 1(a)) the name column is duplicated in both tables, in filtered inheritance (Figure 1(b)) the PackagedIn value is nil for the AbstractNamedEntity EntityA and there is the "Class" identifier column.

Reading & Writing. Once the mapping between the tables and the objects is defined, *i.e.,* the descriptor class is completed, reading and writing objects is straightforward. The following code snippet reads all the FAMIXClass objects from a DB, modifies them and stores them back in the DB.

```
famixClasses := session readManyOf: FAMIXClass.
"the famixClasses objects are modified"
session registerAll: famixClasses.
```

"session" is an object storing the connection with the DB. It is also possible to retrieve only the objects satisfying a given block with:

```
famixClasses := session readManyOf: FAMIXClass
  where: [:each | each isAbstract].
```

When a FAMIXClass is read from the DB, all the classes which have a relationship with it (in our example FAMIXMethod and FAMIXPackage) are retrieved on demand in a transparent way. This means that the message "`readManyOf:`" sent to the session object retrieves only FAMIXClasses, not FAMIXMethods and FAMIXPackages. If we send the getter message "`methods`" or "`packagedIn`" to a FAMIXClass object, the collection of FAMIXMethod objects or the FAMIXPackage object are dynamically read from the DB.

3 The Metabase

The Metabase takes as input a meta-model described in Meta and outputs a GLORP class descriptor, which defines the mapping between the object instances of the meta model, *i.e.,* the model, and the database. Figure 2 shows how the Metabase works.

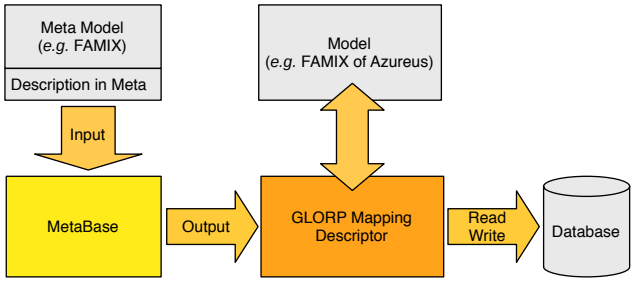


Figure 2. Using the Metabase

Using the Metabase is straightforward. Suppose we have a meta-model described by Meta, *i.e.,* some classes with instance variables described by "meta" methods on the class side. To create the GLORP class descriptor with the Metabase, we can use the following code snippet:

```
classes := OrderedCollection with: ClassA with: ClassB ...
^ClassDBDescriptorGenerator uniqueInstance
  createClassDescriptorForClasses: classes
  named: 'Descriptor' in: aPackage.
```

This code generates the descriptor class named "Descriptor," located in the "aPackage" package. This descriptor can be used to define the mapping between the meta model and the database. To get a connection with the database, which respects the mapping we use the code:

```
db := MetaDB.MetaDBBridge uniqueInstance.
db descriptorClass: Descriptor.
db login: ((Login new)
  username: 'user'; password: 'pass';
  connectString: 'databaseServerLocation';
  database: PostgreSQLPlatform new; yourself).
```

Once the database connection is created, we can create the tables on the database (if the database is empty) with:

```
db createTables
```

and read and write objects of the model with:

```
someClasses := db session readManyOf: ClassA.
someClasses addAll: (db session readManyOf: ClassB).
"someClasses are modified"
db session registerAll: someClasses.
```

Summary. The Metabase generates GLORP descriptors in a fully automatic way. It manages one-to-one, one-to-many and many-to-many relationships among objects. It manages inheritance among meta model classes using filtered inheritance.

4 Example

We present a simple example[6] which shows how the Metabase supports inheritance and all types of relationships (direct, one-to-one, one-to-many, many-to-many).

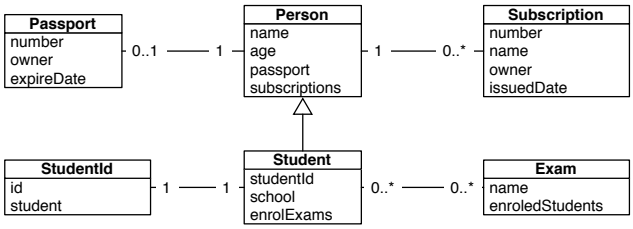


Figure 3. A simplified UML class diagram of the example meta model

Figure 3 shows the UML diagram of the example model. The code snippet below shows the class methods used to describe the meta model in Meta. Due to lack of space we show only the methods for the Person, Subscription, and Passport classes.

```
Person>>metamodelAge
  ^(EMOF.Property name: #age type: Number)
Person>>metamodelName
  ^(EMOF.Property name: #name type: String)
Person>>metamodelPassport
  ^(EMOF.Property name: #passport
    opposite: #owner type: Passport)
Person>>metamodelSubscription
  ^(EMOF.Property name: #subscription opposite: #owner
      type: Subscription multiplicity: #many)
    isDerived: true; yourself.

Subscription>>metamodelIssuedDate
  ^(EMOF.Property name: #issuedDate type: Date)
Subscription>>metamodelName
  ^(EMOF.Property name: #name type: String)
Subscription>>metamodelNumber
  ^(EMOF.Property name: #number type: Number)
Subscription>>metamodelOwner
  ^(EMOF.Property name: #owner
    opposite: #subscription type: Person)

Passport>>metamodelExpireDate
  ^(EMOF.Property name: #expireDate type: Date)
Passport>>metamodelNumber
  ^(EMOF.Property name: #number type: Number)
Passport>>metamodelOwner
  ^(EMOF.Property name: #owner
      opposite: #passport type: Person)
    isDerived: true; yourself.
```

Once the meta description is defined as shown in the code snippet, we can generate the GLORP descriptor, read and write objects instances of the meta model from and to the database as described in the previous Section. Figure 4 shows a screenshot of some database tables automatically generated and populated.

[6]The model of the example can be found in the package "MetaDBTest::ExampleDBs", while the generation of the descriptor is on the class side of the class ClassDBDescriptorGenerator.

Person

dbid	school	studentid	passport	age	name	classtype
1	UZH	2	4	22	Pierazzo	Student
2	USI	3	5	20	Pierone	Student
3	*NULL*	*NULL*	1	27	Marco	Person
4	*NULL*	*NULL*	6	28	Il Puzzone	Person
5	*NULL*	*NULL*	2	31	Jonny Bravo	Person
6	Politecnico	1	3	18	Pierino	Student

PersonExamLink

dbid	personid	examid
1	2	2
2	1	3
3	6	4
4	2	4
5	6	1
6	1	1
7	6	2

StudentId

dbid	id	student
1	1	6
2	3	1
3	2	2

Exam

dbid	name
1	Calculus
2	Algebra
3	Greek
4	Physics

Figure 4. A screenshot of the generated database

5 Conclusions

In this paper we have presented a novel approach to support interoperability in reverse engineering, based on object persistency. Instead of using text files to exchange software system models, we propose the use of the *Metabase*. The Metabase takes a meta model description and automatically generates the object persistency descriptor, *i.e.,* the mapping between the objects (instances of the meta model) and the generated database. We implemented the Metabase on top of the Moose reengineering environment, relying on Meta for the meta model description part and on GLORP for the object persistency part.

Future Work. We have used and tested the Metabase with small and relatively simple meta models (but which already include inheritance and all the types of relationships). We plan to test the Metabase with larger and more complex meta models. We also plan to improve the performance of the Metabase, especially with respect to GLORP proxies and cursors, the bottleneck of the current implementation.

References

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[2] A. Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM Press.

[3] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX: Exchange experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.

NOREX: Distributed collaborative reengineering

Mihai Balint[1], Petru Florin Mihancea[1], Radu Marinescu[1], and Michele Lanza[2]

[1]LOOSE Research Group, Politehnica University of Timişoara, Romania
[2]Faculty of Informatics, University of Lugano, Switzerland

Abstract

Several reengineering environments have been created to provide a unified infrastructure in which various approaches can be employed together. These environments are typically implemented in different languages and use similar yet distinct architectures. While the collaboration between tools is facilitated within such environments, collaboration across their boundaries happens at most at the level of data exchange using files in a specific format, such as XMI and GXL. Consequently, the different groups of researchers are only collaborating shallowly via data, rather than at the level of analysis. In this paper, we present our vision of a distributed service-based reengineering environment that allows different groups of researchers to transparently use and combine existing analysis techniques shifting focus from tool development towards idea development and facilitating the expression of more complex techniques in a fully reusable manner.

Keywords: reengineering, web services, distributed systems

1 Introduction

Reengineering is a complex task that requires several techniques to be employed together [3]. The larger the data at hand, the more techniques we require to manipulate and to reason about the data.

With new reengineering analyses constantly being proposed and implemented, it is highly desirable for researchers to be in a position to use these state-of-the-art techniques, both for comparison of research results, and for reuse of engineering effort. Hence, several environments have been created to provide a unified infrastructure in which various approaches can be employed together [6].

While environments foster collaboration within their boundaries, they are isolated from one another, analyses built in one of them are not reusable in another one. Communication between environments is confined to data transfer via exchange formats [4, 7]. The main cause of this isolation resides in that environments are oftentimes implemented in different programming languages, and even when they are implemented in the same language they have a dedicated infrastructure (*e.g.,* different meta-models, different front-ends).

In this context, we started the *NOREX* project[1], in order to develop an infrastructure that would support research as it unfolds in the current academic and industrial ecosystem. We aim *NOREX* to have following traits: (a) it should be *distributed i.e.,* it should keep the implementation of reengineering techniques near their creators while making them accessible to everyone without the need of reimplementation; (b) it should be *language independent i.e.,* it should allow the writing of techniques in any programming language while enabling their reuse from any particular programming language; (c) it should be *community focused*, meaning that it centers on the user's needs: researchers combine, compare, develop and enhance techniques; reengineers use techniques for analyzing software systems by employing various techniques developed by researchers. Yet, both sides need the convenience of using their familiar environments and programming languages. Furthermore, we are aware that they wouldn't like to invest into integrating different reengineering tools.

2 Collaborative Reengineering

Reengineering research consists of creating new techniques either from scratch or by combining existing ones and it usually involves the creation or extension of the metamodel entities. These extensions and the fact that sometimes the technique is tightly coupled with the metamodel implementation represent the greatest barrier in the face of approach portability.

To overcome this hurdle, research groups have tried to unify metamodels [1] and to create metamodel transformations [2]. However, metamodel transformations are a temporary solution because metamodels evolve as techniques do, and why would we confine researchers to work with a single metamodel?

Although reengineering is unlikely to adopt a common metamodel, we envision a framework that enables researchers

[1]see http://loose.upt.ro/norex

to *choose* the model of software that best suits them is already here. In this context, *NOREX* allows researchers to keep their programming language and their favorite reengineering environment. It also allows them to use others techniques in their own environment and to publish their techniques so that others may use them in their environments.

To achieve this level of flexibility the architecture of today's reengineering tools (see Figure 1) needs several types of adaptation:

- *Decouple the tool's services*. Reducing the coupling between techniques (parsers, analyses) and metamodel implementations ensures that the techniques can be used on different language implementations of the metamodel. For example, decoupling the browser from a particular metamodel will allow it to navigate any metamodel.
- *Specify explicit interfaces*. If each technique provides a structured specification of it's interfaces and of the data structures it uses, it is easier to reuse and understand what it does, hence it is likely that it will get noticed and used by other researchers.
- *Integrate with the community*. *NOREX* implements a means of publishing the services provided by a tool on a NOREX *server* so that they become available for the whole community. It also provides a means of accessing services published elsewhere in the community.

By reusing instead of reimplementing, reengineering is likely to advance faster and our tools are likely to become less interesting as case studies and more interesting as research tools.

3 Current Achievements

Currently, we have implemented a prototype of the *NOREX* reengineering infrastructure, and defined precise methodologies for defining/adapting reengineering techniques in form of sharable *NOREX* services and for using these services. As part of this process, we have created a *NOREX* server implementation and have published several services (*e.g.,* a set of software metrics and metrics-based rules for detecting design flaws related to size and complexity, coupling and inheritance relations, described in [5]). Additionally, we have published more complex analysis, like polymetric software visualization (*e.g.,* the *System Complexity View*[5]). Furthermore, we are in the process of adapting the two analysis environments developed in the past by our research groups (*i.e.,*MOOSE [6] and iPLASMA [5] to make them fully compliant with the *NOREX* infrastructure.

4 Conclusions

We envision that giving researchers the choice of which model of software to use will inevitably lead to the creation

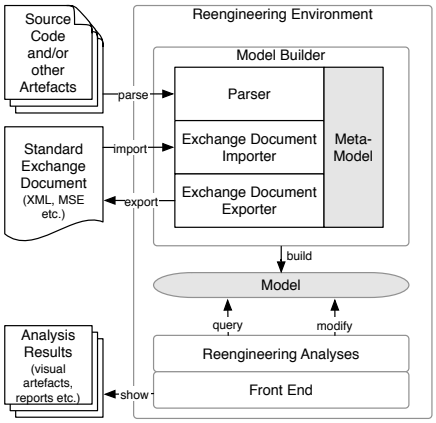


Figure 1. The Architecture of a Generic Reverse Engineering Environment.

of a metamodel market economy, and in time some metamodels will raise as the most useful for certain applications. In this market metamodels will compete in areas like expressiveness, technique richness, community support and popularity. Also a community such as MOOSE community will be able to share their approaches with other researchers thus tightening relations with other research groups.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "NOREX — Network of Reengineering Expertise" (SNF Project IB7320-110997).

References

[1] A. Alvaro, D. Lucrdio, V. C. Garcia, A. F. do Prado, and L. C. Trevelin. Orion-re: A component-based software reengineering environment. In *WCRE*, 2003.

[2] D. Jin and J. R. Cordy. Integrating reverse engineering tools using a service-sharing methodology. In *Proceedings of ICPC'06*. IEEE Computer Society, 2006.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[4] R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE '00*, Nov. 2000.

[5] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.

[6] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10. ACM, 2005. Invited paper.

[7] Xml metadata interchange (xmi), v2.0, 2005. http://www.omg.org/cgi-bin/doc?formal/05-05-01.

Combining Development Environments with Reverse Engineering

David Röthlisberger and Oscar Nierstrasz
– Id: main.tex 8754 2007-05-15 15:12:17Z roethlis –
Software Composition Group
University of Bern, Switzerland
{roethlis, oscar}@iam.unibe.ch

1. Shortcomings of current IDEs

Understanding and maintaining large software systems is a complex and time-consuming yet inevitable challenge. Most systems frequently change and evolve over time to meet new requirements. To perform these changes a software engineer must have an in-depth understanding of the inner structure and implementation of a system. However, the integrated development environment (IDE) usually provides little in the way of support to ease the understanding and changing of software systems.

A large body of research exists in the area of reverse engineering and many promising concepts to improve program comprehension have emerged, such as polymetric views and class blueprints [4]. But the visualizations of reverse-engineered information about software systems are usually separated from the user's working environment, the IDE, and integrated into dedicated reverse engineering tools such as Moose or CodeCrawler [3]. This means that a programmer working on maintenance tasks does not have access to important information about the structure or the dynamic behavior of a software system, such as a view presenting how a class communicates to other classes.

For instance, due to the lack of dynamic information (*e.g.,* collaborators of a class) the developer is forced to frequently browse a large code space without the benefit of goal-directed navigational support. Empirical studies report that an engineer performing maintenance tasks on a system spends at least 35% of the time in navigating dependencies between source artifacts (*e.g.,* which class uses which other classes) . Obviously, current IDEs do not provide adequate means to support the navigation and browsing of software artifacts.

2. Combining Forward and Reverse Engineering within the same IDE

Integrating reverse engineering tools and visualizations into the IDE is one possible way to mitigate the navigational load weighing on a developer, because such visualizations quickly reveal information about the architecture and the structure of a system, hence the user can more efficiently locate important artifacts in a large code space [6].

However, just integrating reverse engineering tools into one single environment is not enough, because there is a mental gap between the goals of reverse engineering and development environments. A developer normally thinks in terms of static entities, *i.e.,* source artifacts such as classes and methods, and analyzes them to get an understanding of the behavior of a system. This metaphor of understanding and developing software is well established and cannot easily be changed. But this metaphor needs to be enriched, for instance we want to quickly get from an abstract feature (*e.g.,* a login feature in a Wiki application) to the developer's view, this is, the source artifacts realizing the feature. For instance, if a developer knows exactly with which other classes a given class dynamically collaborates to realize a specific feature, she can much faster navigate all entities relevant to a certain task, *e.g.,* correcting a defect in that feature [7].

What is needed is not only an integration of ideas, concepts, tools and visualizations gained by reverse engineering into the IDE, but a combination of the conventional, often purely structural view of today's IDEs with a dynamic view on a system and its features. In particular when working with object-oriented languages where polymorphism and late binding is applied (such as Java or Smalltalk) analyzing the dynamic behavior is the most efficient and reliable way to get a complete understanding of a program.

If we manage to tightly integrate dynamic informa-

tion into the IDE and especially into the user's workflow to navigate and browse a software's code space, we may be able to greatly reduce the time required to understand the implementation of a software system, to maintain and evolve it. For short-term adaptations an IDE with integrated reverse engineering is certainly beneficial, but also long-term evolution is easier to achieve within such an IDE, especially if also historical information about a software system are accessible in the IDE [1].

3. Steps to Integrate Reverse Engineering into an IDE

We aim to study the following concrete enhancements and improvements of IDEs for program comprehension, navigation of source artifacts and maintenance of software systems:

- *Related Entities* By dynamically analyzing an application we can for instance reveal how a class collaborates with which other classes. We simply track all the receivers of messages sent from within a class. Making this information accessible helps the programmer to understand the dynamic behavior of a system, because she knows the dynamic collaborators of a class which are difficult to find out by only statically analyzing source artifacts.
- *Object Tracking* When just looking at the static source code we do not know what kind of objects a certain variable (instance or local) will contain when the system is running. By analyzing the system runtime we can memorize the objects that have been stored in a variable and *e.g.,* display the classes of these objects in the IDE while studying the static source code [5].
- *Feature Identification* By identifying source artifacts participating in a specific feature of an application, the developer can more efficiently navigate and also understand all source entities realizing that feature when she has to focus only on a subset of all the source artifacts of the whole system [2, 7].

To actually realize these items we need to perform dynamic analysis. In Smalltalk tools and framework for dynamic analysis are already available, that's why we will develop a prototype of our IDE in Squeak Smalltalk. Even though the low level tools for gathering dynamic information already exist, the challenge is still to integrate and present this information in a useful manner.

This challenge consists of at least four technical issues we need to address: First, to collect dynamic information we have to run the application under study. This means that we have to continuously collect dynamic information on every run of the system to get information as accurate as possible. The second issue is the large amount of data which naturally emerges when performing dynamic analysis. This forces us to focus on a small percentage of available information that has proven to be useful and to filter out all other information. The third issue is to find good techniques to visualize and present the dynamic information dependent on the context the developer is in. A fourth issue we need to address is how to store and update the dynamic information accurately. In order to be useful the information has to be displayed without letting the user wait and it has to be accurate, so old information from an previous run of the systems needs to be invalidated.

A last but yet important step is the validation of the resulting IDE. For a serious validation of our work the developers need to be involved, this means that we will perform several empirical studies to report on the usefulness and the added value of combining reverse engineering with development environments.

References

[1] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[2] O. Greevy and S. Ducasse. Characterizing the functional roles of classes and methods by analyzing feature traces. In *Proceedings of WOOR 2005 (6th International Workshop on Object-Oriented Reengineering)*, July 2005.

[3] M. Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[4] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[5] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[6] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, Nov. 2004.

[7] D. Röthlisberger, O. Greevy, and A. Lienhard. Feature-centric Environment. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft 2007) (tool demonstration)*, 2007.

Scripting Diagrams with EyeSee

Matthias Junker, Markus Hofstetter
Software Composition Group, University of Bern, Switzerland

Abstract

Presenting numbers in the right way is crucial for understanding their meaning. We present EyeSee, a diagram drawing engine that allows for programatic specification of the presentation, while offering default values that produce uncluttered diagrams.

1 Introduction

Using diagrams to reason about quantitative data is a common practice in today's age of information. However, reading and understanding diagrams is not always as easy as it could be. In many cases, embellishments capture more attention than data [1]. In this paper, we present EyeSee, a diagram engine that aims to provide default values that produce uncluttered diagrams. In particular, EyeSee focuses on a simple motto proposes by Tufte: *Minimize chart-junk and maximize data ink* [2].

Furthermore, most diagram drawing tools are concentrated on the user interface, and typically they require input data in a certain format. With EyeSee we address the researcher that builds the analysis, and that is comfortable with his programming environment. Thus, EyeSee does not require the data to be passed in a fixed format, but rather it allows the user to specify programatically how to extract the data from the model he is using.

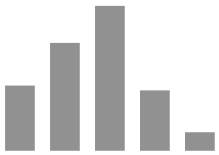
While on the one hand we strive to produce clutter-free diagrams, on the other hand, we also focused on the ability of the user to control all the details of the presentation with as little effort as possible. For this purpose we provide scripting methods that allow for changing most of the properties of a diagram (e.g., the color of the elements, the type of axis, the size).

Our prototype is built in Smalltalk and works with Smalltalk domain models, but the same approach can be applied to any programming language.

2 EyeSee by example

In this section we show the basic facilities of EyeSee using hands-on examples. As domain model, we use a collection called *contributors* containing the contributors to a certain project, as obtained from a versioning system. Every contributor has the attributes name, number of commits, lines of code, team and versions.

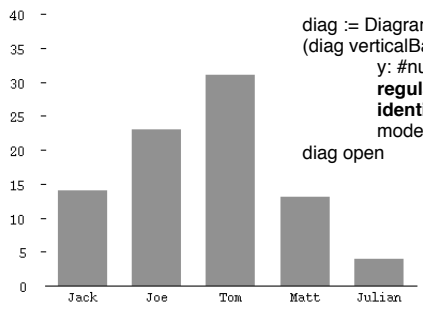
We start with an very simple script that opens a vertical bar diagram where each bar shows the number of commits of the contributor. In the script, *numberOfCommits* is a method in the Contributor class:



```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
        y: #numberOfCommits;
        models: contributors.
diag open
```

In the above diagram we cannot see which bar is from which contributor, and we also do not know the actual values of the bars. Thus, we want to add axis and labels to the bars.

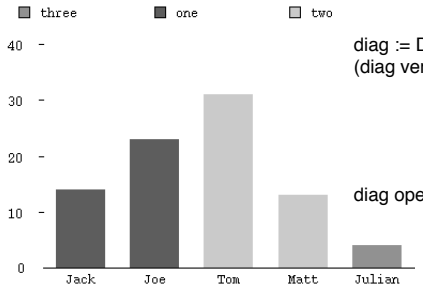
We accomplish this with 2 additional lines of code. The new line *regularAxis* adds axis with ticks and labels. In regular axis, the distance between two ticks is always the same. Instead, we could also use *valueAxis*, which draws ticks dependent on the values which get displayed in the diagram. For putting a label below every bar, we can use the *identifier* keyword, to tell the diagram, how it can get the data for the labels. In our example, we use the *name* of the contributor to identify the bars (where *name* is a method in the Contributor class):




```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
        y: #numberOfCommits;
        regularAxis;
        identifier: #name;
        models: contributors.
diag open
```

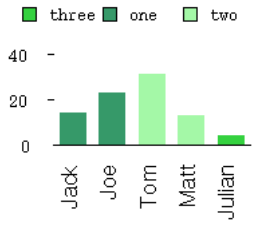
We might also want to encode other information in our diagram. We have this possibility by using colors or shades

of gray for the bars. In our example, knowing for each contributor the team he belongs to, we also want to reveal this information. Thus, we add one line to specifying that we want a different color for each new *team*.




```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
        y: #numberOfCommits;
        identifier: #name;
        colors: #team;
        regularAxis;
        models: contributors.
diag open
```

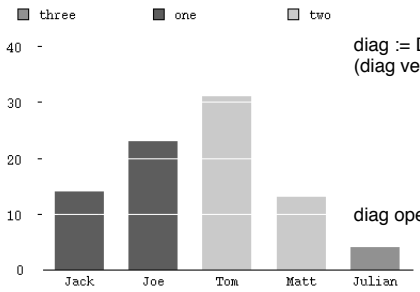
By default, the values for the color get encoded with gray scale colors. By adding the expression *lightColors* or *strongColors*, we can use a set of ten strong or light Colors which are distinct enough so they can be easily separated [3]. However, custom colors can also be used for the encoding, by using the *useColors:* message:




```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
        identifier: #name;
        y: #numberOfCommits;
        color: #team;
        useColors: #(
            seaGreen
            limeGreen
            paleGreen)
        width: 200;
        height: 200;
        regularAxis;
        models: self model.
diag open
```

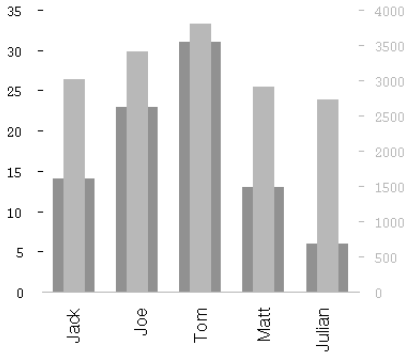
In the above example, not only did we specify the colors we want to use, but we also specify the width and height of the diagram. In our example, as there is not enough space for the identifiers to be displayed horizontally, they automatically get rotated and are displayed vertically. Of course we can also configure to have them rotated manually by using the *rotatedIdentifiers* message.

It is not always easy to estimate the values displayed in a diagram, especially when the axis with the values are far away from the actual data elements. The usual approach is to add a grid to the diagram, typically with a gray color. As opposed to that, we provide grid lines that are only showed on top of the bars with slight white stripes (as proposed by Tufte [2]).

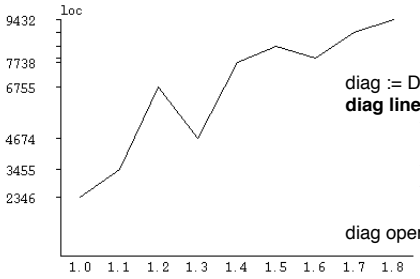



```
diag := DiagramRenderer new.
(diag verticalBarDiagram)
        identifier: #name;
        y: #numberOfCommits;
        color: #team;
        regularAxis;
        gridLine;
        models: self model.
diag open
```

In EyeSee we can also compose multiple diagrams in every possible combination. For example, below we show combine the previous bar diagram with another bar diagram that shows how many lines of code each contributor has written (the code is not shown due to space constraints).



EyeSee offers several types of diagrams. For example, the next one shows a line diagram. In our example, we want a line diagram that shows the change of number of lines of code in a project during a year.




```
diag := DiagramRenderer new.
diag lineDiagram
        y: #loc;
        identifier: #versionNumber;
        yAxisLabel: 'loc';
        valueAxis;
        models: self versions.
diag open.
```

3 Internals of EyeSee

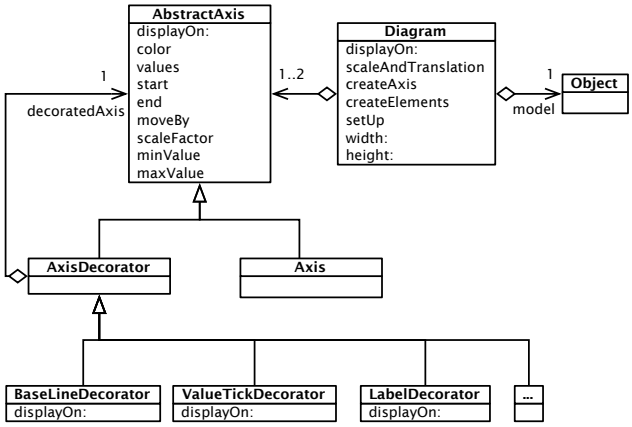


Diagram. The *AbstractDiagram* class is the root of our diagram hierarchy. It contains the model of the data and the logic that creates the painting of the diagram. The model can be set using the `models:` message. We assume this to be a collection of the model objects that the user is working with. To let the diagram know how to extract the data from the model object we use the `y:` message. The parameter is a block that, when sent to one of the model objects will let it answer the relevant value. It is mandatory to set those two attributes if the user wants to create a meaningful diagram. For the other attributes of the diagrams like we provide reasonable default values. Once those two things are done we can call the `setup` method to properly initialize the axis and the elements of a diagram. The elements of a diagram encapsulate the graphical representation of the values with the corresponding model object. For example an element in a bar diagram consists of a rectangleShape (the bar), the color we want to draw this bar in and the model object that belongs to this bar. If an element receives the `displayOn:` message, it simply forwards it to its shape. The possibility to return the underlying model object allows for some interactivity.

Axis. The *Axis* class contains the collection of values that are relevant to this axis (e.g. the y-axis of a horizontal bar diagram knows about the height of the bars) and some other properties like the color. But an axis has no knowledge of the other parts of the diagram (diagram and axis).

To allow for the axis of the diagrams to be displayed in various manners according to the needs of the user, we implemented a decorator pattern. Every axis can be wrapped in several decorators. The decorators and the axis are subclasses of *AbstractAxis* and every decorator knows the axis/decorator it decorates. If a decorator receives the `displayOn:` message it forwards it to the axis/decorator and then displays itself. This leads to a chain of responsibility where every decorator only draws the parts of the axis which it is responsible for. We provide some basic decorators (*e.g., RegularTickDecorator, ValueTickDecorator, LabelDecorator, BaseLineDecorator etc.*) that can be composed by the user.

The benefit of this pattern is that the displaying behavior of the axis can be changed depending on the used decorators. For example we could show Ticks on the Axis in regular intervals or by changing the decorator we could show them only if there is a corresponding value in the diagram. This also allows for extensibility because if there is a new idea on how the axis should be drawn the only thing that needs to be done is the implementation of a decorator that responds to the `displayOn:` message in the desired way.

4 Future work

To make the customization of diagrams a bit easier we plan to build a graphical user interface for EyeSee. This benefits the user because he does not have to memorize or look up the name of the scripting commands. Instead he can modify the properties of a diagram using checkboxes, sliders.

Another direction we want to advance EyeSeein is interaction between diagrams. We consider to implement a tool that allows the user to create several diagrams and script the interaction between them. For example he could script a scatterplot and a bar diagram. If he selects a point in the scatterplot the corresponding bar in the bar diagram would be highlighted. Or maybe he wants to display a popup containing a small histogram when he moves the mouse over a property to see how it is distributed. The user would be able to script such interactions directly in the tool.

5 Conclusion

The goal of *EyeSee* was to create a diagram drawing engine that lets the user visualize his data without forcing him to convert it into a fixed format. The provided default values produce clear and clutterfree diagrams. We implemented scripting methods to make it easy to create and customize diagrams. The scripts are small and hide the internals of our model.

References

[1] S. Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[2] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

[3] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

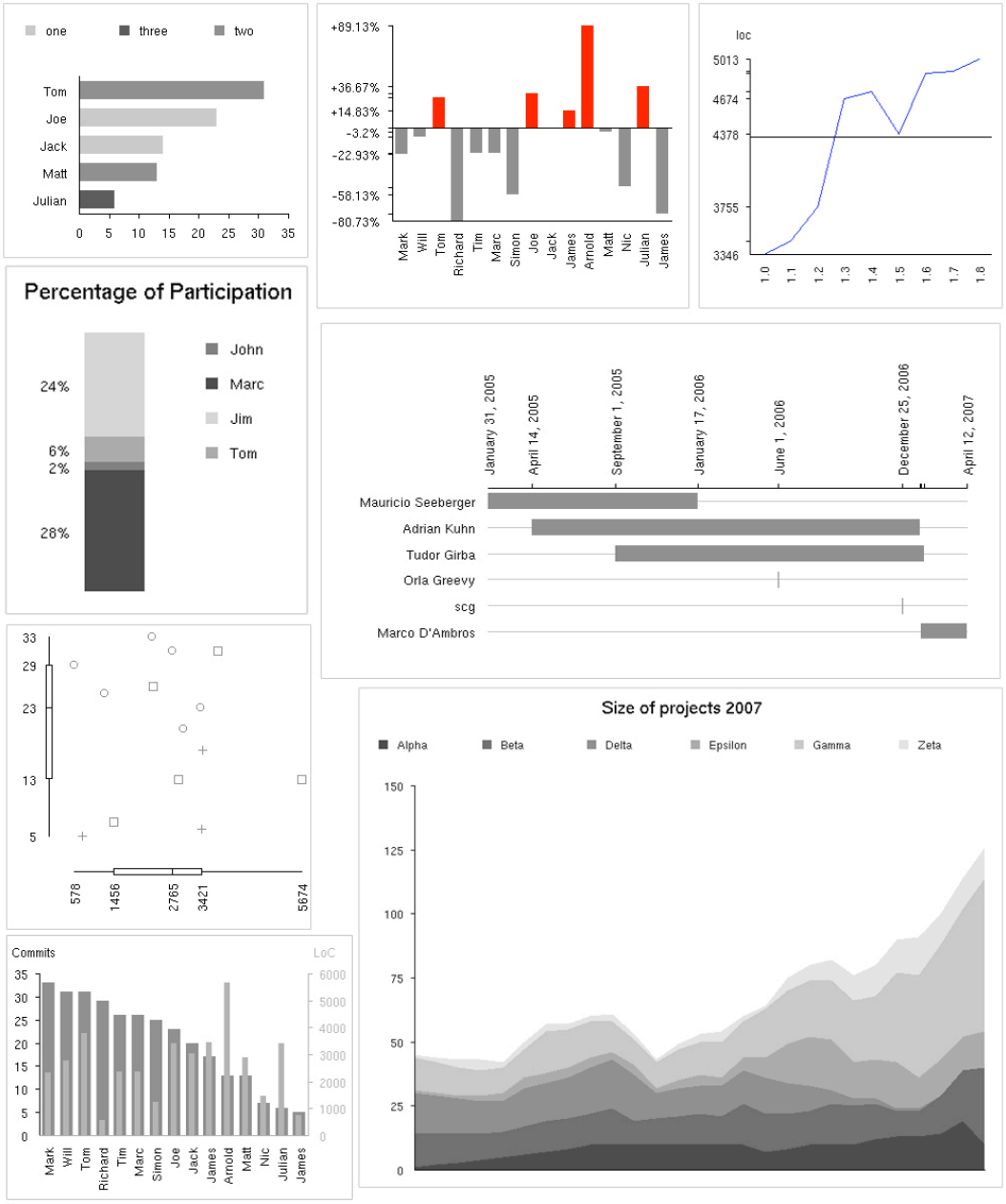


Figure 1. Several diagrams supported by EyeSee

Package Surface Blueprint: A Software Map

Stéphane Ducasse* Damien Pollet Mathieu Suen Hani Abdeen Ilham Alloui
Language and Software Evolution Group — Université de Savoie, France

Abstract

Large object-oriented applications are structured over large number of packages. Packages are important but complex structural entities that may be difficult to understand since they play different development roles (i.e., class containers, code ownership basic structure, architectural elements...). Maintainers of large applications face the problem of understanding how packages are structured in general and how they relate to each others. In this paper, we present a compact visualization, named Package Surface Blueprint, that qualifies the relationships that a package has with its neighbours. A Package Surface Blueprint represents packages around the notion of package surfaces: groups of relationships according to the packages they refer to. We present two specific views one stressing the references made by a package and another showing the inheritance structure of a package. We applied the visualization on two large case studies: ArgoUML and Squeak.

This paper makes heavy use of colors in the figures. Please obtain and read an online (colored) version of this paper to better understand the ideas presented in this paper.

1 Introduction

To cope with the complexity of large software systems, applications are structured in subsystems or packages. It is now frequent to have large object-oriented applications structured over large number of packages. Ideally, packages should keep as less coupling and as much cohesion as possible [2], but as systems inevitably become more complex, their modular structure must be maintained. It is thus useful to understand the concrete organization of packages and their relationships. Packages are important but complex structural entities that can be difficult to understand since they play different development roles (*i.e.*, class containers, code ownership basic structure, architectural elements...). Packages provide or require services. They may play core role or contain accessory code features. Maintainers of large applications face the problem of understanding how packages are structured in general and how packages are in relation with each others in their provider/consumer roles. In addition, approaches that support application remodularization [1, 6, 7] succeed in producing alternative views for system refactorings, but proposed changes remain difficult to understand and assess. There is a good support for the algorithmic parts but little support to understand their results. Hence it is difficult to assess the multiple solutions.

Several previous works provide information on packages and their relationships, by visualizing software artifacts, metrics, their structure or their evolution [5, 3]. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance relationships of the internal classes, how the internal class inherit from external ones,...) and help identifying their roles within an application.

In this paper, we propose Package Surface Blueprint, a compact visualization revealing package structure and relationships. A package blueprint is structured around the concept of surface, which represents the relationships between the observed package and its provider packages. The Package Surface Blueprint reveals the overall size and internal complexity of a package, as well as its relation with other packages, by showing the distribution of references to classes within and outside the observed package. We applied the Package Surface Blueprint to a large case studies namely Squeak the open-source Smalltalk comprising more than 2000 classes.

Section 2 presents the challenges that exist to support package understanding, it also summarizes the properties that a visualization should satisfy to be effective. Section 3 presents the structuring principles of a package blueprint which are then declined to support a reference view and an inheritance view in subsequent sections. In sections 4, we discuss our visualization and position it w.r.t. related work before concluding.

*We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).

2 Challenges in Understanding Packages

We present a coarse list of useful information to understand packages. Our goal here is to identify the challenges that maintainers are facing and not to define a list of all the problems that a particular solution should tackle.

Size. What is the general size of a package in terms of classes, inheritance definition, internal and external class references, imports, exports to other packages? For example, do we have only a few classes communicating with the rest of the system?

Cohesion and coupling. Transforming an application will follow natural boundaries defined by coupling and cohesion [2]. Assessing these properties is then important.

Central vs. Peripheral. Two correlated pieces of information are important: (1) whether a package belongs to the core of an application or if it is more peripheral, and (2) whether a package provides or uses functionality.

In addition, packages reflect several organizations: they are units of code deployment, units of code ownership, can encode team structure, architecture and stratification. Good packages should be self-contained, or only have a few clear dependencies to other packages [2, 4]. A package can interact with other ones in several ways: either as a provider, or as a consumer or both. In addition a package may have either a lot of references to other packages or only a couple of them.

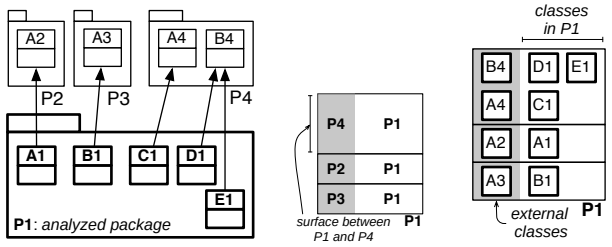
3 Package Surface Blueprints

A package blueprint represents how the package under analysis references other packages. Figure 1 presents the key principles of a Package Blueprint. These principles will be realized slightly differently when showing direct class references or inheritance relationships.

3.1 Basic Principles

The package blueprint visualization is structured around the "contact areas" between packages, that we name *surfaces*. A *surface* represents the conceptual interaction between the observed package and another package. In Figure 1 (a) the package P1 is in relation with three packages P1, P2, and P4, via different relationships between its own classes and the classes present in the other packages, so it has three surfaces.

A package blueprint shows the observed package as a rectangle which is vertically subdivided by each of the package's surfaces. Each subdivision represents a surface between the observed package and a referenced package, and will be more or less tall, depending on the strength of the relation



(a) Analyzed package (b) Stacked surfaces (c) With classes

Figure 1. Consider P1 that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). Small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part) (c).

between the two packages. In Figure 1, the package blueprint of P1 is made from three stacked boxes because P1 references three other packages. The box of the surface between P1 and P4 is taller because P1 references more classes in P4 than in P2 or P3.

In each subdivision, we show the classes involved in the corresponding surface. By convention, we *always* show the classes in the referenced packages in the leftmost gray-colored column of each surface, and the classes of the observed package on the right. In Figure 1, the topmost surface shows that class A1 references class A2. If many classes reference the same external class, we show them all in an horizontal row; we can thus assess the importance of an external class by looking at how many classes there is in the row: in Figure 1, the row of B4 stands out because the two referring classes D1 and E1 make it wider.

3.2 Detailed Explanation

To convey more information, we add variations to the basic layout described above, as illustrated in Figure 2.

Internal references. To support the understanding of references between classes inside the observed package, we add a particular (red bordered) surface at the top of the blueprint. We name this surface the head of the blueprint and the rest its body. Since this surface displays internal references, its left gray column for external classes will remain empty. In addition, the first column represents the internal classes that are referenced from within the package itself: here A1 and G1 are the classes referenced respectively by B1 and C1 and H1 and I1. The height of the red surface indicates the number of classes referenced within the package.

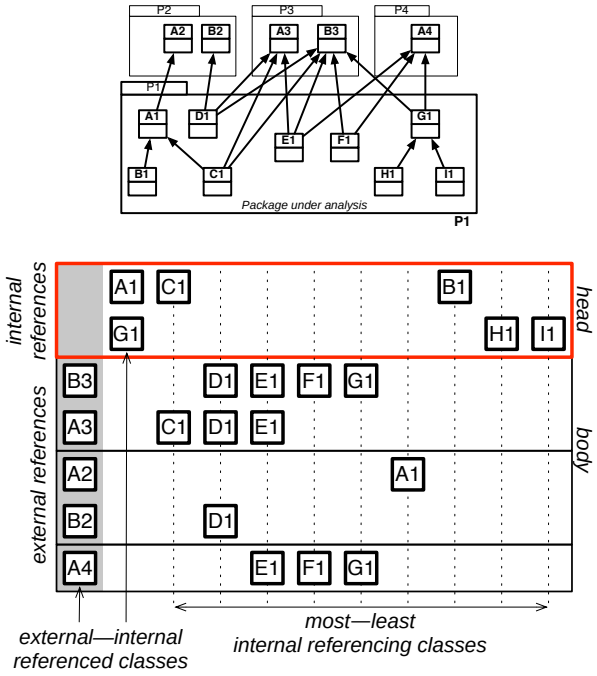


Figure 2. Surface package blueprint detailed view.

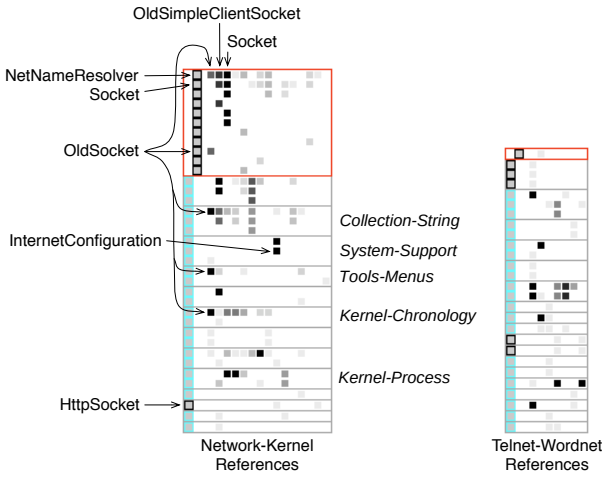


Figure 3. Analysing the Network-Kernel Package.

Position. Internal classes are arranged by columns: each column (after the first one of the red surface) refers to the same internal class for all the surfaces. The width of the surface indicates the number of referencing classes of the package. Figure 2 shows that class C1 internally references A1, and externally references A3 and B3.

We order classes in both horizontal and vertical direction to present important elements according to the (occidental) reading direction. In the horizontal direction, we sort classes from left to right according to the number of *external* classes they reference from the whole package. Hence classes referencing the most occupy the leftmost columns in the white area of the package blueprint.

We apply the same principle for the vertical ordering, both of surfaces within a blueprint, and of rows (*i.e.*, external classes) within a surface. Within a package, we position surfaces that reference the most classes the highest. Within a surface, we order external classes from the most referenced at the top, to the least referenced are at the bottom of the surface. This is why in Figure 2 the surface with P3 is the highest and why the surface with P2 is above P4 (since there are more classes references from P2 than from P4.

Color. We want to distinguish referenced classes depending on whether they belong to a framework or the base system, or are within the scope of the application under study. When a referenced class is not part of the application we are currently analyzing, we color its border in cyan. In addition the color intensity of a node conveys the number of references it is doing: the darker the more references. Both intensity and horizontal position represent the number of references, but position is computed relative to the whole package, while intensity is relative to each surface. Thus, while classes on the left of surfaces will generally tend to be dark, a class that makes many references in the whole package but few in a particular surface will stand up in this surface since it will be light grey.

3.3 An Example: The Network Subsystem

We are now ready to have a deeper look at an example. The Squeak Network subsystem contains 178 classes and 26 packages — this package contains on the one hand a library and a set of applications such as a complete mail reader. The blueprint on the left in Figure 3 shows the references package blueprint of the Network-Kernel package in Squeak.

Glancing at it we see that the package blueprint of the Network-Kernel package has nearly a square top-red surface indicating that most internal classes are referenced internally. This conveys a first impression of the package's cohesion even if not really precise [2]. Contrast it with the package blueprint of the Telnet-Wordnet package which clearly shows little internal references.

We see that Network-Kernel is in relation with thirteen other packages. Most of the referenced classes are cyan, which means that they are not part of the network subsystem. What is striking is that all except one of the referenced classes are classes outside the application (see (HTTPSocket) in Figure 3). However, since the package is named *kernel*,

it is strange that it refers to other classes from the same application, and especially only one. We see that half of the referred packages have strong references (indicated by their dark color).

Using the mouse and pointing at the box shows using a fly-by-help the class and package names (indicated in italics in Figure 3). The Tools-Menus surface indicates some improper layering. Indeed it shows that Network-Kernel is referencing UI classes via the package Tools-Menus which seems inappropriate. We learn that the class making the most internal references is named OldSocket; this same class also makes the most external references, to three packages (Collection-String, Tools-Menus, and Kernel-Chronology). The second most referencing class is named OldSimpleClientSocket. It is worth to notice that OldSocket is only referencing itself and that even OldSimpleClientSocket does not refer to it, so it could be removed from this package without problems. The third most referencing class is Socket. Having two classes named Socket and OldSocket clearly indicates that the package is in a transition phase where a new implementation has been supplanting an old one.

4 Evaluation

The Package Surface Blueprint shows the internal number of classes as well as the number of classes externally referenced. Hence it conveys whether the package is using a lot of information.

Size. The Package Surface Blueprint shows the complexity of the observed package in several dimensions. The height of the body indicates the amount of external classes referenced, whereas the number of surfaces shows the number of referenced packages. The height of each individual surface shows how many classes are referenced int the corresponding package. This gives us an estimate of the coupling with the package of this surface; to further evaluate the coupling strength, we should also look at the intensity of referencing classes in the surface because it represents the number of references.

Central or Peripheral. By looking at the border color of external classes (cyan or black), we can easily see if a package depends a lot on the platform or on the application. Also, by using the selection mechanism, we can interactively see if a package is imported by different subsystems (central) or just by specific ones (peripheral).

Cohesion and Coupling. The package blueprint also makes it possible to roughly compare how several packages are coupled with the observed one: larger surfaces indicate coupling to more classes and are positioned nearer to the head surface, while surfaces with more darker class squares represent packages which are more coupled in term of sheer number of references.

5 Conclusion

In this paper, we tackled the problem of understanding the details of package relationships. We described the Package Surface Blueprint, a visual approach for understanding package relationships. While designing Package Surface Blueprint, we tried to exploit gestalt visualization principles and preattentive processing.

We successfully applied the visualization to several large applications and we have been able to point out badly designed packages. To help interpretation, we have identified a list of recurrent striking blueprint shapes. We also introduced interactivity to help the user focus and navigate within the system. We were however rather knowledgeable about both the visualization and the studied systems; in future work, we plan to validate the package blueprint usability tests with several independant software maintainers.

References

[1] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 1999.

[2] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[3] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 70–77. IEEE Computer Society, 2005.

[4] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[5] M. Lungu, M. Lanza, and T. Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[6] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[7] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

Package References Distribution Fingerprint

Hani Abdeen Ilham Alloui Stéphane Ducasse* Damien Pollet Mathieu Suen
Language and Software Evolution Group — Université de Savoie, France

Abstract

Object-oriented languages such as Java, Smalltalk, C++ structure their programs using packages which allow classes to be organized into named abstractions. Understanding how packages are structured and how they relate to each others is a very important task during the maintenance of large aplications. Cohesion and coupling metrics are still among the most used metrics during the perfective maintenance (they help to point to candidate packages for restructuring), but they do not help the maintainers to understand the packages' structure and interrelationships. In this paper, we present an approach of pre-attentive 2D visualizations, named Reference Distribution Matrix. We present also its use for generating two views that help to understand the relationships that a package has within its system. We applied the visualizations on three large Java case studies: JBoss, Azureus, and ArgoUML.

1 Introduction

To cope with the complexity of large object-oriented software and to facilitate their maintenance, classes are partitioned into packages. The organization of classes is usually following their conceptual interrelation, but as systems evolve, their modular structure must be maintained. It is thus useful to understand the concrete organization of packages and their relationships. Ideally, packages should keep as less coupled and as much cohesive as possible [2]. According to different overviews of package design [8, 6, 1, 7], we find two approaches of design principles. First are those that relate the cohesion of a package to the interconnections between its internal classes. Second are those that relate the cohesion of a package to the use of its internal classes within the system (*i.e.,* if two classes of a package are used from a common client subsystem, they are thus conceptually related, regardless of the explicit relationships that exist between them). We believe that the last approach is closer to the principle of Package (*i.e.*, Module concept in object-oriented languages). Many metrics of package cohesion have been defined [2, 6, 1, 7]. They help to point to candidates packages for restructuring during perfective maintenance. Both approaches do not help maintainers of large applications when they face the problem of understanding how packages are structured in general and how packages are in relation with each others in their provider/consumer roles.

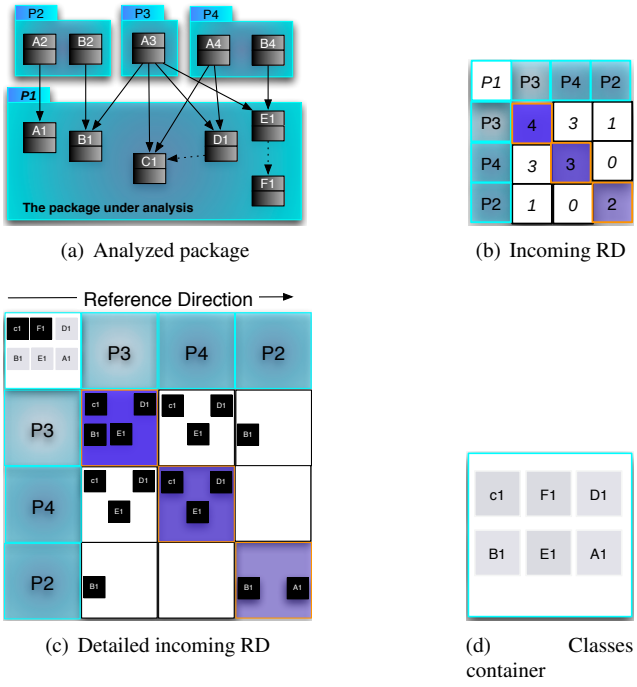
Several previous works provide information on packages and their relationships, by visualizing software artifacts, metrics, their structure or their evolution [5, 3]. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the packages' structures, their interrelationships within the system, and help identifying their roles within an application.

In this paper, we propose Reference Distribution Matrix (RDM), an approach to create a compact visualization revealing package structure and its relationships with the system. Visualizations created using RDM are named by convention RD Fingerprint. A RD Fingerprint is structured around the distribution of references over the internal classes and the related packages. We say that a package (P1) refers to an another (P2) if P1 includes a class (C1) that invokes some methods of an another class (C2) packaged into P2. In the same context, we say that P1 refers to C2, C1 refers to P2, P2 exports C2 for P1, and P2 is referenced by C1/P1.

The RD Fingerprint reveals the overall size of a package in terms of imported packages and classes importer (Outgoing RD Fingerprint); and exported classes, the packages which import it (Incoming RD Fingerprint). In other words, the Outgoing RD Fingerprint shows the distribution of the system over the package's classes, while the Incoming RD Fingerprint shows the distribution of the package's classes over the system. The second view highlights particularly the cohesion, as defined in [7], of the analyzed package. The aim of a RD Fingerprint is to present the information that maintainers need to understand packages, estimate theirs principal qualities and detecting the design quality problems. Usually the useful information which are needed to understand packages structure and roles within system are:

Size. What is the general size of a package in terms of classes, internal and external class references, imports, exports to other packages?

*We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).



(a) Analyzed package

(b) Incoming RD

(c) Detailed incoming RD

(d) Classes container

Figure 1. The basic layout of the Incoming RD Fingerprint.

Cohesion and coupling. Good package should includes classes which are usually needed from the system to perform the same task (*i.e.*, conceptually coupled classes) [7], and have a few clear dependencies to other packages [2]. These properties are among the most used ones during the perfective maintenance[6, 1, 8].

Section 2 presents the RD Fingerprint and its basic principles. In Section 3 we evaluate our visualization via an example before concluding.

2 Package Reference Fingerprints

A RD Fingerprint is given by two views of the analyzed package. First is the Incoming RD Fingerprint that represents how the package under analysis is used by the system. Second is the Outgoing RD Fingerprint that shows how the package uses the remainder of the system. Figure 1 presents the key principles of a Incoming RD Fingerprint. The Outgoing RD Fingerprint keeps these principles with just one variation that we will describe later.

2.1 Basic Principles: Package RDM

The RDM is based on five key principles. We represent them in order of creating an Incoming RDM for a package.

Size. The size of the Incoming RDM for a given package imported by N packages into system, is 1 + N.

Reference Direction. The reference direction is always from left to right. Thus the referencing items are placed on the left, while the referenced ones are on the right Figure 1 (c)

Borders. Referencing packages are placed on the first column starting from the second line and symmetrically on the first line.

Principal Diagonal. Each cell in the principal diagonal represents the group of classes which are referenced by the corresponding package in the first column.

Symmetric Partitions. Each cell that does not belong to the principal diagonal nor to the borders represents the group of classes which are referenced by both the corresponding package in the first column and the corresponding package in the first line:

$$|c(i,j)| = |c(i,1)| intersection : |c(1,j)| \quad (1)$$

In Figure 1 (a) the package P1 is used by three packages P2, P3, P4, so its Incoming RDM size is 4. The packages P2, P3, and P4 refer respectively two, four, and three classes into P1. The class B1 is referenced by P2, P3; the classes C1, D1, and E1 are referenced by P3, and P4; while just the classes C1 and F1 are referenced by internal classes into P1 Figure 1 (c).

For Package Outgoing RD Fingerprint these key principles are the same, but according to the reference direction principe, the first column (i.e. the left side border) should be placed as a right side border (i.e. referenced packages are placed in the last column).

To convey more information, we add variations to the basic layout described above.

Position. Related packages form the visualization borders: top border and side border. According to the reference direction principle, the side border is placed on the left of the Incoming RD Fingerprint, and on the right of the Outgoing RD Fingerprint. The order of related packages is the key to render the visualization meaningful. Therefore we have implemented an algorithm to sort the related packages respecting the following principles: for the groups of classes placed in the principal diagonal, the more related the closer. One contract has been added to this principle: the order of related packages is given from top to down following that the package which has the bigger number of related classes is the higher.

Classes' container and saving place. For harnessing the pre-attentive process [4] into our visualization we have used the saving place principle. After creating the RDM for an

analyzed package, the internal classes which will appear within the RD Fingerprint are arranged as an ordered table. A class container is a visual way to show a cluster of classes through that table: for each class which belongs to the visualized cluster the corresponded place becomes dark; the other places still light.

Color. Classes are always black, while packages are blue. The intensity of packages' color could vary (*i.e.*, from blue to light sky blue) relatively to the degree of coupling between the related package and the analyzed one: The higher coupled, the darker. The border color of visualized packages is turquoise instead of stub packages (i.e. packages that does not belongs to the analyzed system) their border color is gold.

The color of internal cells is produced from two mixed colors: slate blue and orchid colors. The density of the slate blue color equals to the number of classes included into the concerned cell divided by the number of those included into the corresponding cell in the same line and belongs to the principal diagonal. The degree of the blend color (orchid) equals to the number of classes included into the concerned cell divided by the number of all visualized classes (*e.g.*, in an Incoming RD Fingerprint is the number of all exported classes). For example, in Figure 1 (c), known that P1 exports 5 classes Figure 1 (a), the color of *cell(2,3)*, that contains one class, which presents the intersection between classes groups given by *cell(2,2)*, that contains 4 classes, and *cell(3,3)* is produced as following:
(slate blue density: (1/4)) blendWith: (orchid density: (1/5))
In concrete words, this approach help to detect using colors properties where some classes are conceptually coupled and the degree of their coupling, and to detect in the same time, the portion of these classes relatively to the whole visualized classes.

Internal information. Internal information about the references between classes inside the analyzed package are visualized in the first cell of the principal diagonal (e.g. *cell(1, 1)* in Figure 1 (c)). While RD Fingerprint focuses on the relationships between the observed package and the system, it gives also a short overview of the internal side of the package. The classes which have internal dependencies are visualized in the first cell of the principal diagonal. The color of this cell is either green (more than the half of the internal classes have some internal dependencies) or red in the another case. However, we could replace this by a metric.

3 Examples and Evaluation

For evaluating our visualization we have applied it on three large Java case studies: JBoss, Azureus, and ArgoUML. In this section we present the use of RD Fingerprint for analyzing packages via examples.

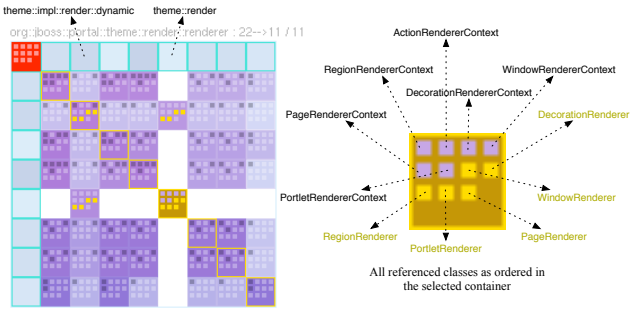
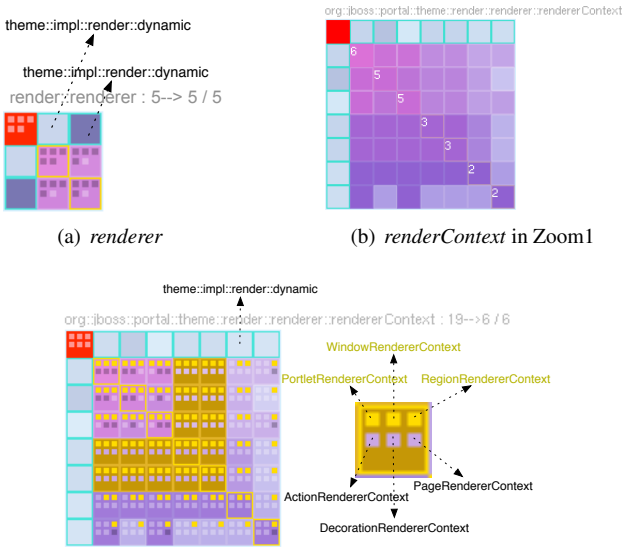


Figure 2. Analysing *them.render.renderer* package within *JBoss* application.

In Figure 2 we visualize the Incoming RD Fingerprint for the package *them.render.renderer* taken from JBoss application. First, the width (or the height) of the view that this package is not referenced from a lot of another ones, and referencing classes are not equally distributed between referencing packages (*i.e.*, some packages in the border sides are darker than some another). For example the package *them.impl.render.dynamic* is darker than the package *them.render*, which means that the first is more coupled with the analyzed package. The second information that one could note rapidly that the first cell on the principal diagonal is red and there is no dark place visualized inside it. Thus we know that the internal classes do not have dependencies between them. While the size of the classes container (*i.e.*, the size of any cell) shows that an important number of classes has some incoming references. In two words, *them.render.renderer* exports 11 classes to the system. By looking inside the view we see that the internal classes are not really coupled because the texture of the matrix is not correlated. Some classes in *them.render.renderer* are something coupled because they are referenced together from the same packages. That is what we see when some cells form together a local correlated texture (*i.e.*, a region having cells with the same color). In the other hand, the group of classes placed in the cell(6, 6) has a lack of cohesion. Clicking on the container of this group, we select it and all its classes and we note that they are referenced by just two packages: *them.impl.render.dynamic* and *them.render*. The last refers to all of them (5 classes) while the first refers to 4 classes of them and two other classes.

Inspecting these classes allowed us to know that they are the classes responsible of items rendering(*PageRendering*, *WindowRendering*, etc.). While the other classes packaged in *them.render.renderer* are those which are responsible of items rendering context (*PageRenderingContext*, *WindowRenderingContext*, etc.). Thus we took the decision to re-modularise these classes by moving the classes of rendering context to a new package



(a) *renderer* (b) *renderContext* in Zoom1

(c) *renderContext* in Zoom2

Figure 3. *them.render.renderer* **package after the factorization**

them.render.renderer.rendererContext. The result of this factorization is given in Figure 3 which shows that we have now two cohesive packages: *them.render.renderer* and *them.render.renderer.rendererContext*. The first is highly cohesive because its texture is fully correlated and its color is orchid (*i.e.*, all referenced classes are coupled). While the second is cohesive with some classes (*RegionRendererContext*, *WindowRendererContext* and *PortletRendererContext*) that are highly coupled (*i.e.*, they are referenced together by most of the client packages Figure 3). The unique 'bad' result of this factorization that the package *them.impl.render.dynamic* becomes coupled with *them.render.renderer* and *them.render.renderer.rendererContext*, while before it was coupled just with the first.

As result, we could say that RD Fingerprint gives us, within a glance, at minimum five different information about the analyzed package: (1) the size of referencing packages, (2) the size of referenced classes, (3) the density of internal references, (4) coupling degree for the referencing packages, and (5) the cohesion of the analyzed package and the coupling between its internal classes. In addition, using classes container and the selection mechanism of a group of classes help to detect rapidly the coupling of classes and if they are referenced always together, if they are sometimes coupled with other classes, or some of them are more coupled than the remainder Figure 3 (c). One another hand, it allows us to use a *Zoom* mechanism and shows the cohesiveness of a package without the need of visualizing its classes Figure 3 (b).

4 Conclusion

In this paper, we tackled the problem of understanding the details of package relationships. We described the RDM, and its use for providing Incoming RD Fingerprint as a visual approach for understanding package relationships and visualizing the package cohesion and classes coupling. While designing RD Fingerprint, we tried to exploit at maximum pre-attentive processing using color properties and saving place principle. We also introduced interactivity and multi-selection mechanism to help the user during the analysis task.

We successfully applied the visualization to several large applications and we have been able quickly to point out badly designed packages.

In future work, we plan to use RDM for creating other views and to validate our viusalization usability tests with several independant software maintainers.

References

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In *CSMR, Lisbon*, 2001.

[2] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[3] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 70–77. IEEE Computer Society, 2005.

[4] C. G. Healey, K. S. Booth, and E. J. T. Harnessing preattentive processes for multivariate data visualization. In *GI '93: Proceedings of Graphics Interface '93*, 1993.

[5] M. Lungu, M. Lanza, and T. Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[6] H. Melton and E. Tempero. The crss metric for package design quality. In *ACSC '07*, 2007.

[7] L. Ponisio and O. Nierstrasz. Using contextual information to assess package cohesion. Technical Report IAM-06-002, University of Berne, Institute of Applied Mathematics and Computer Sciences, 2006.

[8] L. Rising and F. W. Calliss. Problems with determining package cohesion and coupling. *Software - Practice and Experience*, 22(7):553–571, 1992.

Reverse Engineering through Holistic Software Exploration

Mircea Lungu *and* Michele Lanza
Faculty of Informatics - University of Lugano, Switzerland

Abstract

Software Exploration tools usually work at a single level of abstraction. We argue for an approach which integrates multiple levels of abstraction in exploration. Each of these levels presents complementary information which is useful for the reverse engineering and understanding process.

Introduction. Software exploration is a technique to analyze software projects. Most software exploration tools work at a single level of abstraction. We propose the concept of *holistic software exploration* which encourages exploring multiple levels of abstraction at the same time.

We distinguish the following abstraction levels that contain useful information about software projects:

- The super-project level, i.e., the context or contexts in which a project exists. We are not aware of research in exploration at this level.
- The project macro level, i.e., the modules composing a system and the interactions between them
- The project micro level, i.e., fine-grained entities such as classes and methods

Super-Project Level. Software projects are *social animals*. They do not exist alone but in the context of other projects. Companies and research groups use versioning repositories in which their projects cohabitate. In the extreme case, super-repositories such as SourceForge can also be considered as the ultimate context of the many open-source projects hosted on it. Sometimes the context can be useful for learning important facts about the system's eco-system, i.e., its environment.

The screenshot Figure 2 presents one application to perform exploration at super-project level. The application is *The Small Project Observatory* [1]. The conventions are:

- The projects have different colors and the graphics are stacked one on top of the other.
- The time is divided in equal intervals (*e.g.,* months)

1 See http://www.inf.unisi.ch/lungu/observatory

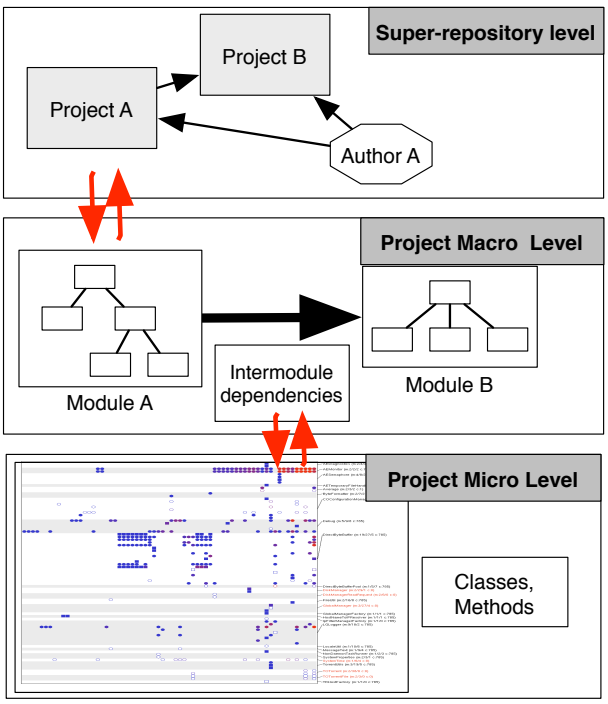


Figure 1. The Concept of Vertical Software Exploration

- The height of the intervals reflects the size and respectively activity of the projects in terms of classes and commits per month
- If the projects are not changed during an interval the color of the interval is 50% transparent

If one looks only at each project as an individual (see Figure 2), he can observe that after the middle of 2005 the size of the bottom project did not change. However, if one looks at information in the whole context of the project, one can see that there is a correlation between the time the bottom project stops and the top project starts to grow. Another observation is that the activity of the blue project stops and the activity of the brown one starts. A new repository was started, with the contents of the old project, while the old

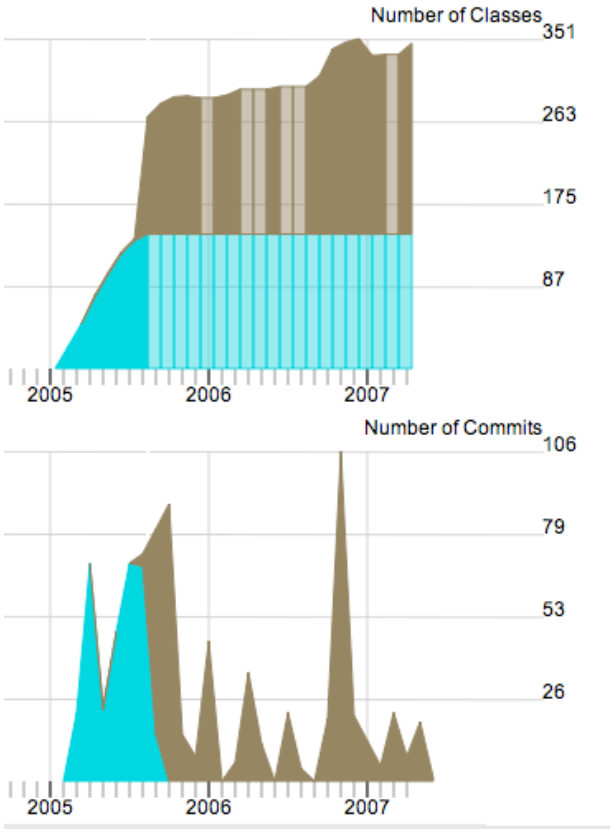


Figure 2. The evolution of size and activity for all the projects in the Bern Store which have 'lungu' as author.

one remained in place.

We extract other types of information, such as author collaborations, inter-project code sharing, information about inter-project dependencies, etc.

Project Macro Level. The super-repository level is useful up to a point. After that, the reverse engineer has to focus the analysis at a lower abstraction level. We consider the next level to be the one where modules and their interdependencies are presented. Softwarenaut [4, 3, 2] (see Figure 3), our module interdependency exploration tool is built on top of Moose[1].

Usually at this level the projects are decomposed hierarchically and one can explore the hierarchy.

Project Micro Level. The next level of abstraction is the level of classes and methods. There are various ways in which one can navigate to this level coming from the previous one. One is to explore a certain module in depth, disconnecting it from the rest of the system. Another one is to explore a relationship between two modules, such as we

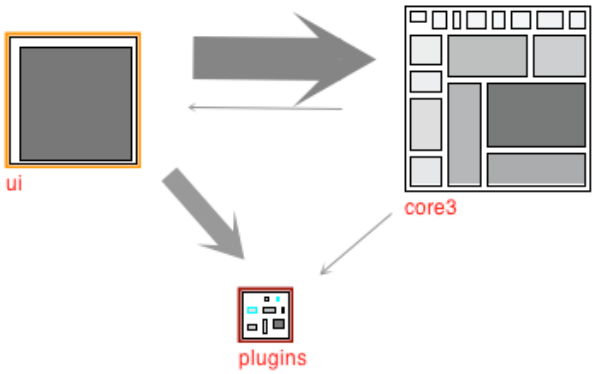


Figure 3. The dependencies between three top-level modules in Azureus

did in our work on intra-dependency analysis [3] (see Figure 4).

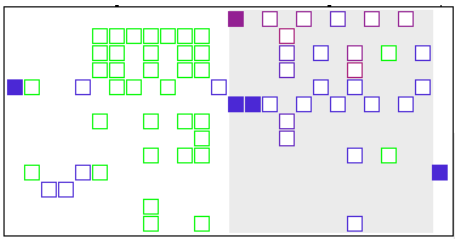


Figure 4. The Semantic Dependency Matrix is an intra-dependency visualization technique

References

[1] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[2] M. Lungu, A. Kuhn, T. Gîrba, and M. Lanza. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 95–100, 2005.

[3] M. Lungu and M. Lanza. Softwarenaut: Cutting edge visualization. In *Proceedings of Softvis 2006 (3rd International ACM Symposium on Software Visualization)*, pages 179–180. ACM Press, 2006.

[4] M. Lungu and M. Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006*, pages 349–350, Los Alamitos CA, 2006. IEEE Press.

Graph Theory for Software Remodularisation

Ilham Alloui, Stéphane Ducasse
Language and Software Evolution Group — Université de Savoie, France

Abstract

Software system modularisation consists in a partitioning a software system into a set of components given a set of criteria/features (structural, behavioural, ...). Software re-modularisation as part of a reengineering process aims at producing new system modularisations for a given purpose: reuse, maintenance or evolution. Many approaches have been proposed based on entity clustering, graph dependencies, formal concept analysis and on matrices. Those approaches generally succeed in providing system re-organisation but they mostly lack in providing users (i.e., maintainers) with support to evaluate remodularisation alternatives, in terms of impact on the quality criteria defined for the system. Our posiion, advocated in this position paper, is to go a step further than those approaches by helping users take decisions through a more advanced remodularisation support. Our approach is based on both source code metamodel and the graph theory.

1 Introduction

Large software applications are hard to master and need tool support for understanding their architecture and organisation. In particular, learning the structure of code developed by others is especially time and effort intensive and need to be supported by appropriate tools [5]. In such a context, visualisation is necessary but not sufficient. Perfective or corrective maintenance requires further support, in particular for system remodularisation, that provides them with alternatives together with their impact on target quality criteria (maintainability, cohesion, coupling, etc.). Visualisation must then be used in conjunction with other program understanding techniques such as software metrics [7], static and dynamic source code analyses [8].

Many approaches have been proposed based on entity clustering, graph dependencies, formal concept analysis and on matrices[9, 4, 12, 1, 10]. They generally succeed in providing system re-organisation but they mostly lack in providing users (*i.e.*, maintainers) with support to evaluate remodularisation alternatives. One challenging issue is to go a step further than those approaches by helping users take decisions through a more advanced support. Investigating the graph theory is promising in this context as it incorporates a set of algorithms that could be adapted to remodularisation support.

The remainder of this paper is organised as follows. In Section 2, we present our approach to software remodularisaton support. Before concluding, we discuss in Section 3 some challenging issues we are addressing in our ongoing work.

2 Graph theory for supporting software remodularisation

In this section, we describe our approach of remodularisation support, founded on both source code metamodel [6] and graph based algorithms. Source code meta-models such as FAMIX support application analysis of source code written in multiple languages. Graph based algorithms are chosen to benefit from a well established theory that could be applied and extended/customised to many purposes. As shown in figure Figure 1, remodularisation support provides, on the one hand, a set of tools whose role is to propose to the users system re-organisation alternatives (the question here is "'where and how can we change the system to better source code reorganisation?"'), and on the other hand, tools that help those users evaluate the impact on defined quality criteria (to answer the question: "'what are risks and benefits if I choose this alternative?"'). In particular, such a support should help evaluate to what extent, each alternative modifies the system behavior, its complexity, reusability or its conceptual architecture. Other quality criteria related to the maintenance process could be evaluated as well: cost, effort, etc.

3 Discussion

Package organisation may reflect several design criteria, among them:

- functionality usage (librairies/frameworks)
- functionality specialisation/extension

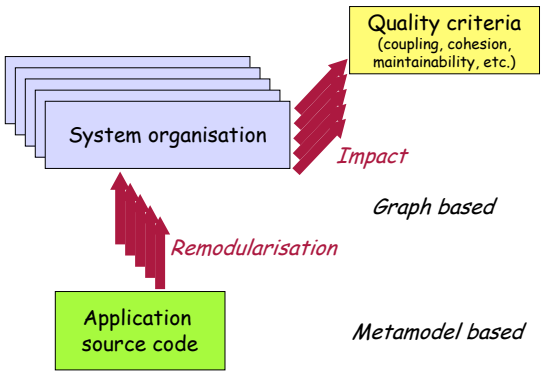


Figure 1. Remodularisation Support

- team organization

Challenging issues we address:

Characterising packages. In particular their main functionality and roles. Packages may play different roles (*e.g.*, library, executable application, core framework to be extended/specialised, etc.). This information is not always available to maintainers and depending on the designers, it could include more than one design criteria (as packages are long-lived entities, constinuously under change by possibly many developers). Taking into account those aspects, proposals for remodularisation could be provided to both better identify package functionaly and ease package use/reuse. Such information could be extracted from grouping classes to reveal package functionality [10, 1].

Splitting entities (classes/packages). Most legacy systems have been designed apart from nowadays design principles like reuse, extensibilty, etc. Application source code generally lacks modularity and encapsulation. One challenging issue is to provide support for (re)modularising such applications so that future extension/reuse is eased. Another case is large packages could be divided into thiner ones based on grouping class functionalities [2, 12, 4].

Breaking dependencies. References among classes/packages often lead to unexpected dependencies. Those dependencies often should be reconsidered and some could be revoked if they are irrelevant, *e.g.*, usage dependency from a UI toward an application core class. Another case is that of classes that are are strongly mutually consumer/provider to each other. Ideally relationships between classes should be entirely one way to limit the impact of change to one class instead of both.

Handling cycles. Bad design or successive versions produced by different people could lead to cyclic references. Remodularisation support help handling cycles for instance through breaking dependencies [11].

Evaluating remodularisation impact on quality criteria. Among criteria we are interested in: cohesion, coupling and maintainability [3]. As quality criteria could be conflicting such as usability (better UI) and security (restricted access), users should be provided with means for evaluating risks and benefits of proposed alternatives [13]. Connected components from the graph theory could for instance be used to evaluate the impact of a change on structurally connected entities.

4 Concluding Remarks

In this position paper, we address the problem of relevant support to software remodularisation. We highlight the need for an advanced support that not only provides maintainers with remodularisation alternatives but helps them evaluate the impact of each alternative on quality criteria expected for the system. We propose to use graph theory to support re-modularisation effort.

References

[1] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 1999.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of 18th Conference on Automated Software Engineering (ASE'03)*, pages 267–270. IEEE Computer Society, Oct. 2003. Short paper.

[3] L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[4] E. Burd and M. Munro. Evaluating the use of dominance trees for c and cobol. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 401, Washington, DC, USA, 1999. IEEE Computer Society.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering

environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[7] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[8] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, Sept. 1999. IEEE Computer Society Press.

[9] M. Saeed, O. Maqbool, H. A. Babri, S. Z. Hassan, and S. M. Sarwar. Software clustering techniques and the use of combined algorithm. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 301, Washington, DC, USA, 2003. IEEE Computer Society.

[10] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[11] D. Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[12] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 33–43. IEEE Computer Society Press, 1997.

[13] S. Wohlfarth and M. Riebisch. Evaluating alternatives for architecture-oriented refactoring. In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 73–79, Washington, DC, USA, 2006. IEEE Computer Society.

Surgical Information to Detect Design Problems with MOOSE

Muhammad Usman BHATTI[1], Stéphane DUCASSE[2]
[1]*CRI, Université de Paris 1 Sorbonne, France*
[2]*Language and Software Evolution Group, Université de Savoie, France*

muhammad.bhatti@malix.univ-paris1.fr, stephane.ducasse@univ-savoie.fr

Abstract

The quality attributes, such as understandability and modularity manifest their importance in the later part of the software life where a lot of resources are required to maintain or reuse software whose quality has been marred by the urgencies of time to market. In this position paper, we present and analyze an existing system meant to be reused on various product lines. We intend to use the MOOSE framework to precisely identify the needs of a reengineer in terms of code-smells, visualization and metrics. In this position paper, we discuss some of the limitations of the existing software system and inadequacy of existing toolkits to automate the task of detection of these limitations. We intend to discuss the appropriateness of MOOSE as a remedy to these deficiencies.

1. Introduction

Software design is an iterative process and it is very difficult to achieve an ideal design in terms of quality in the first iteration. While managers and developers are working to satisfy the marketing needs in a minimum time, software quality is their last preoccupation. Therefore, the use of a certain technology or a paradigm doesn't mean ideal software quality. Reverse Engineering and Reengineering are used to analyze the code of a software having very little or no existing documentation [1]. One of the aims of reegnineering activity is to generate high-level models and diagrams from the only artefact that represents the true state of the affairs vis-à-vis business of an enterprise, along with the technical architecture to support this business, so that these can be visualized and manipulated by reengineers.

The first author is working in an company which produces a range of blood plasma analysis automatons. The overall cost of an instrument includes a huge chunk for software that drives the instruments and managers are always looking to reduce the cost of development, and once developed, the cost of software maintenance. Development costs can be reduced by reusing the existing artefacts which are developed earlier to be reused in the existing development process.

But the existing software contains a lot of deficiencies in terms of the quality of its components partly due to the lack of knowledge of object-oriented design methodologies.

We would like to use a tool-based, reverse-engineering approach in order to improve the quality of existing software to enable the reuse of individual components. For that purpose, MOOSE [7] seems to be an ideal candidate with its visualization and metric-based plug-ins. One of the limitations that has been encountered during the work is the absence of the research tools supporting Microsoft .NET Framework and associated languages. For this purpose, we are in the process of development of a plug-in to integrate MOOSE with .NET environment.

2 Case Study: Blood Plasma Analysis Machines

We are working in a company that builds blood plasma analysis machines. The machine is composed of two main parts: the hardware part concerning mechanics and electronics like the arms, the drawers, and the software part managing the hardware with an aim of analyzing the blood plasma. The user of the machine (operator, biologist, etc), after authentication, loads one or more tubes of plasma, as well as products, in the drawers of the machine, associates a test of analysis each tube, and launches the analysis. The automaton performs the analysis for blood-related diseases and the results of the analyses are displayed on the display device.

For the sake of precision and clarity, we shall only be talking about the layer that manages the business objects and operates with the database layer to manage the data associated with these objects. This is one of the sub-systems of the software that is replicated on each new machine.

Although the software cannot be considered to be a legacy one, it presents rudimentary examples of object-oriented code lacking object-oriented design [4]. We discuss this issue in detail in the next section.

3 Business Entity Layer

The business entity layer of the software is supposed to support functionality such as patient data, tubes, blood analyses, results, reagents used, etc. An extract for the class diagram of this layer is shown in the figure below.

Figure 1. Class Diagram Entity Layer

Although the above diagram is not very clear in terms of its contents, nevertheless it communicates some facts about the business entity layer.

- There is a lack of hierarchical structure, via inheritance - one of the tenet of code reuse and a fundamental element of object-oriented design.
- Presence of huge classes encapsulating functionality pertaining to multifaceted objects and entities.

These problems which are visible from a very high-level abstract extraction from the code in question, show that rudimentary restructuring is needed in terms of design and quality of the software.

Unfortunately, most of the tools available for C# language provide this level of view and reengineers are supposed to manually follow the track from these abstract maps. These visualizations are helpful in understanding the overall system architecture but not adapted to fine-grained problems associated with the software architecture. Argument stating that software metrics only represent numbers and provide information that require further interpretation, in this case, seems to carry weight [2]. They do not go farther than the information depicted in the figure above since they do not provide comprehensive information to ease the task of software reengineering. For example, Table 1 is a set of quality metrics for the mammoth class in Figure 1 above, called Servicepatient.

Table 1. Metrics for Servicepatient Class

Metric	Value
Lines of Code	4000
# of Methods	262
Depth of inheritance	1
Lack of Cohesion Of Methods (LCOM)	0.8501908
LCOM Henderson-Sellers (LCOMHS)	0.8534483
Cyclomatic Complexity	1368

For a reengineer, the gap between the visualization needed for her refactoring activity and information provided by the metrics remains largely obscure.

An example may illustrate the things better: While manually analyzing the software in question we found that there is no object related to the patient's tube and the logic to add, search, manipulate and delete the tube data associated to tubes is scattered in other classes of the system. These classes directly operate over the table tube found in the database, and we used manual techniques to identify places where the information was changed in the database as shown in Figure 2. During code inspection, we found that any change to the tube related logic requires changes in several classes spread over two software layers. These are clear code smells for encapsulation related problems [5]

This bad code smell related to the absence of encapsulation could be identified using visualization techniques [6], it is not straightforward since most visualization techniques do not support the detection of this design anomaly [8, 3]. In addition, it requires a manual effort to understand the effects of absence of this object on the future evolutions of the software.

The extract class refactoring suggested by [5] provides a possible solution consisting of extracting all the methods and attributes related to tubes to be extracted from the main class. This extracted code can then be placed in a class that can encapsulate the logic associated to the tube object and the code can be modified accordingly. The visualization has a great part to play to support unwanted effects of the refactoring exercise. This can be done by identifying all the attributes and methods that are effected by the movement of the code from one class to another. As defined above, the visualization techniques support more legacy code problems such as code duplication and large classes. But there is no visualization technique that supports the actual refactoring process.

A reengineer at this point would find the following information useful:

- Classes containing the logic pertaining to a particular object, in our case tube.

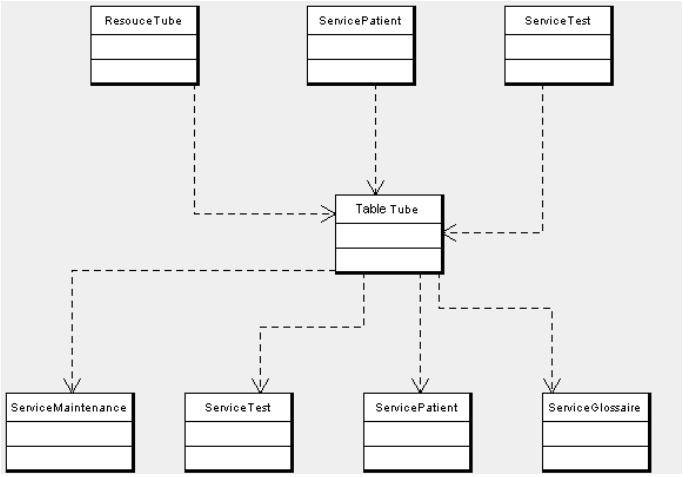


Figure 2. Classes directly manipulating the Patient's Tube Information

- Dependencies of these classes at a coarse-grained level and the methods at a fine-grained level to calculate the impact of restructuring activities on other modules.
- A set of metrics indicating the impact on software quality with the progress of software restructuring.

For this purpose, we intend to exploit the MOOSE system to extract what we call surgical information to guide our reengineering activities. The purpose of this exercise is to find a set of visualizations and metrics that identify problems more precisely than the existing techniques. For this purpose, more details and semantically rich metrics such as FAN in analysis and Formal Concept Analysis should be investigated. This would allow the reengineer to gain an insight into the code which provides much more than mere structural information to detect design defects.

4 Tool Used - Related Work

The number of research studies and tools for C# based systems is inferior to that of Java-based systems due partially to the license and availability issues of the two products. Nevertheless, some research prototypes are still available to study existing C# systems. For this activity, we used Altova's class diagram extractor to extract the class diagrams for the module in question. NDepend tool provides a useful tools with limited license for academic purposes to calculate various metrics of the software systems developed in C#. These include an extensive set of information for the dependencies of type-based information. But since tube type doesn't already exist in the system, it already requires a lexical analysis of the system to search for possible candidates methods and properties named tubes. This is the limitation of this tool that the reengineer needs to resort to other tools for lexical analysis of various statements to study the dependency of a software concern deeply embedded in the system. Devmetrics provides a set of high-level metrics for the system developed in C# language but doesn't provide class diagram extraction functionality. Doxygen is useful tool to generate Javadoc-type documentation for the existing systems. It can be easily inferred that existing applications for the reengineering of C# systems lack all-in-one information needed to guide the reengineering activity.

5 Conclusion

Software quality is understood to be a topic of academic interest in the industrial world. The software generally have inferior quality due to time to market urgencies. Hence, new software is needed to be developed for each new product without reusing the existing components. We are currently working on a system of this type and trying to ameliorate its quality. We are facing a lack of tools which may guide reengineer's activities. Normally a mix of tools are used to extract this information. In addition, C#-based systems are not studied in research for the problem of restructuring, hence the lack of tools and information is accentuated for this family of systems. We intend to use MOOSE framework to infer what we call surgical information needed to guide reengineering tasks with a set of visualizations and metrics.

References

[1] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[2] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 18, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Y. Crespo, C. López, R. Marticorena, and E. Manso. Language independent metrics support towards refactoring inference, jul 2005. http://pisuerga.inf.ubu.es/clopez/refactoring/.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[7] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of moose: an agile reengineering environment. volume 30, pages 1–10, New York, NY, USA, 2005. ACM Press.

[8] C. Parnin and C. Görg. Lightweight visualizations for inspecting code smells. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 171–172, New York, NY, USA, 2006. ACM Press.