JFREEDOM: a Reverse Engineering Tool to Recover Framework Design

Nuno Flores
FEUP, Universidade do Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
nuno.flores@fe.up.pt

Ademar Aguiar
Universidade do Porto, INESC Porto
Rua Dr. Roberto Frias s/n
4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt

ABSTRACT

Object-oriented frameworks are a powerful technique for large-scale software development capable of delivering very high levels of design and code reuse. To be effective, software engineers must invest a considerable amount of time on understanding and learning how to reuse a framework. To understand a framework is usually difficult due to the design being very complex and hard to communicate, especially when not properly documented. This paper presents a semi-automated reverse engineering tool, based on Eclipse, to assist on the recovery of framework design, and thus to help on the understanding and evolution of frameworks. The tool, under development, uses a multi-phased approach that recovers design elements of increasing level of abstraction, ranging from hooks and templates to meta-patterns and design patterns.

Keywords

Object-oriented frameworks, design patterns, reverse engineering

1. INTRODUCTION

The increasing reliance on information technology for consumer and industrial goods imposes new requirements on software flexibility. The major trends in customer requirements are [26]: customer specific modifications and software versions (custom made systems), much faster response to change requests and new requirements (evolution), and the ability to easily modify the software based on the immediate customer needs (tailoring).

In most cases, addressing a solution to these problems is prone to deviations due to tight development schedules, inappropriate development process, insufficient resources and weak design decisions.

Despite these constraints, solutions are developed resorting to the well-accepted object-oriented approach, which was often promoted as one of the most effective approaches to build reusable, flexible software and quickly adopted by the industry throughout the last couple of decades.

While the benefits of object-oriented development were widely recognized, its application did not necessarily produce the expected results. A new generation of inflexible large application systems arose from the improper use of object-oriented techniques, fueled by the short time-to-market and improper software evolution and maintenance.

Typically, the result covers the immediate needs and, sometimes, it also concerns with maintenance and evolution issues.

The existing know-how of the development team may be quickly diluted over time due to inactivity, or when members leave to other projects. Lack of proper documentation, large amounts of code and a poor and complex design makes it hard for a newcomer to find its way through the system. When confronted with new requirements or new projects, it may be not clear to the software engineer where to act upon the existing system without running the risk of damaging it.

Nevertheless, the system often lacks clarity about its reusability and flexibility.

In order to satisfy new requirements, software engineers often need to reengineer existing object-oriented software systems, ranging from monolithic legacy systems to complex frameworks and their components, into more flexible systems easier to reuse.

As one of the most complex kinds of object-oriented products, frameworks are thus very difficult to reengineer, especially by software engineers not initially involved in its original design. The difficulty is typically due to its design complexity (abstractness, incompleteness, high flexibility, and obscurity) and lack of proper documentation.

This paper presents JFREEDOM, a reverse engineering tool to assist on the recovery of framework design, and therefore to help also on framework reengineering. JFREEDOM is delivered as an Eclipse plugin and supports a semi-automated design recovery process.

The tool under development uses a multi-phased approach to recover design elements of increasing level of abstraction,

ranging from hooks and templates to meta-patterns and design patterns.

After an overview of the key design elements of frameworks, the paper describes the reverse-engineering process followed by the tool and the major modules of the tool illustrated by an example. The paper concludes with an outline of the work being carried on this research and discusses our approach in relation to other existing approaches.

2. FRAMEWORK DESIGN

The definitions for a framework are not consensual and vary from author to author [1]. Shortly, a framework can be defined as a semi-complete design and implementation for a set of applications in a given problem domain.

Frameworks are firmly in the middle of the reuse techniques: they are *more abstract and flexible* (and harder to learn) than components, but *more concrete and easier to reuse* than a raw design (although less flexible and less likely to be applicable). Frameworks are considered a powerful reuse technique because they lead to one of the most important kinds of reuse, the reuse of design.

Typically, the design of a framework is very complex [8] because it is:

- very abstract, to factor out commonality;
- incomplete, requiring additional classes to create a working application;
- more flexible than the strictly needed by the application at hands;
- and obscure, in the sense that it usually hides existing dependencies and interactions between classes.

The main concern of framework design is to separate the aspects that are invariant along several applications in a domain — *the frozen spots* — from the other domain aspects that vary among applications and thus must be kept flexible and customizable — *the hot spots*, also called *variation points* [13, 23].

The key elements used to design a framework can be divided in three levels of abstraction: *template-hook mechanisms*, *meta-patterns*, and *design patterns*.

2.1 Template and Hook methods

Frameworks provide their flexibility at hot spots using two essential constructs: *templates* and *hooks*. Template and hook methods [14, 22, 23, 32] are two kinds of methods extensively used in object-oriented programming and, in particular, in the implementation of frameworks.

Template methods are implemented based on hook methods, and call at least another method. A hook method is an elementary method in the context where the particular hook is used, and can be either an abstract method, a regular method, or another template method.

Generally, template methods are used to implement the frozen spots of a framework, and hook methods are used to implement the hot spots. Opposite to the latter, the frozen spots are aspects that are invariant along several applications in a domain, possibly representing abstract behavior, generic flow of control, or common object relationships.

The difficulty of good framework design resides exactly on the identification of the appropriate hot spots that provide the best level of flexibility required by framework users. More hot spots offer more flexibility but it also means having a framework more difficult to design and use, so somewhere in between resides a balanced design.

2.2 Meta-Patterns

The possible ways of composing template and hook classes in the hot spots of a framework were enumerated and presented in [23, 24] under the form of a set of design patterns, called meta-patterns.

Meta-patterns are a useful abstraction that can be applied to categorize and describe framework hot spots on a meta-level.

Template and hook methods can be organized in several ways. Although they can be unified in a single class, in most of the situations, it is better to put frozen spots and hot spots into separate classes. When using separate classes, the class that contains the hook method(s) is considered the *hook class* of the class containing the corresponding template method(s) — the *template class*. We can consider that hook classes parameterize the corresponding template class. The hook methods on which a template method is based can also be organized in different ways. They can be defined all in the same class, or in separate classes, in a superclass or subclass of the template class, or in any other class.

In [23], a clear distinction between the level of abstraction of design patterns and meta-patterns is made. Design patterns describe the design of specific frameworks on an abstraction level higher than the underlying programming language, albeit they provide no means of capturing the design independently of a more or less specific framework example. By using meta-patterns, the design is captured at an higher level, independently of its particular application. Meta-patterns express how the required flexibility — represented by the hot spots — is gained in a particular framework.

Furthermore, seven composition meta-patterns were identified that relate templates with hooks, as depicted in Figure 1. These repeatedly occur in object-oriented frameworks and, thus, in its constituent design patterns.

2.3 Design Patterns

Frameworks and design patterns are concepts closely related, representing two different categories of high-level design abstractions [18]. A single framework typically encompasses several design patterns. Patterns provide an intermediate level of abstraction between the application level and the level of classes and objects.

A design pattern is commonly defined as a generic solution to a recurring design problem that might arise in a given

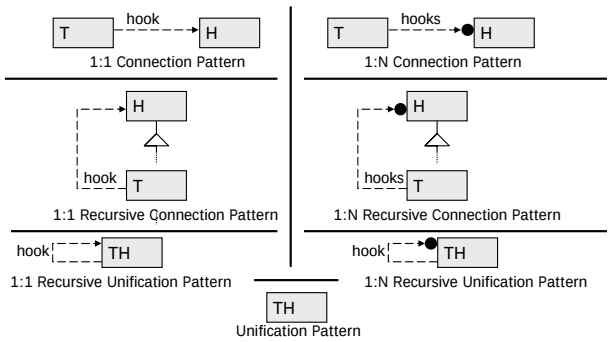

Figure 1: Catalog of meta-patterns.

context [4, 7, 15]. Design patterns are extremely useful to provide the flexibility and extensibility requirements needed at hot spots.

Design patterns can be viewed as a valuable means to document existing frameworks and develop new reusable object-oriented software architectures [23]. Design patterns are particularly good to document frameworks because they capture design experience and enclose meta-knowledge about how the flexibility was incorporated.

The combined use of frameworks with patterns and components is very effective, significantly helping to increase software quality and reduce development effort [10].

Although meta-patterns can be directly used to document the roles of framework participants, they are much more useful to document the roles of the participants involved in a design pattern. Generally, it is preferred to attach meta-patterns to design patterns, and design patterns to concrete participants that instantiate them, instead of attaching meta-patterns directly to concrete participants, mainly due to the redundancy introduced and the level of detail being too fine to be useful.

3. DESIGN RECOVERY APPROACH

The high-level reverse engineering process we devised for framework design recovery starts from the source code, be it a snippet, a module, package, component or an entire framework, and progressively evolves, step by step, until reaching the desired detail of information about the framework design.

In the path to reverse engineer from source code to design patterns, the process performs several consecutive steps, each providing deliverable concrete results, such as: templates, hooks, hot spots, and instances of meta-patterns and design patterns. Each step of the process is fed with the results from the preceding step, delivering itself new results at an higher level of abstraction, following a Pipes & Filters architectural style.

The approach being followed relies on probing the source code for variation points and to successively group them into structures of increasing level of abstraction. Firstly, template and hook methods are identified and grouped into hot spots and meta-patterns, which are then used to identify instances of design patterns with some level of certainty, based on a knowledge base of known design patterns.

3.1 Meta-patterns-based approach

The work in progress stated in this paper thrives to reach three goals:

1. To adopt an approach based on meta-patterns, where each step of the process delivers clear and relevant information about the possible reusability of an object-oriented software system.

2. To use a representation for design patterns based on its composition of meta-patterns. It should be generic enough to be able to represent any design pattern, whether already discovered or yet to unfold.

3. To provide automation and tool support, by developing a tool to automate the reverse engineering process, flexible, interactive and capable of visually deliver its results to the user.

As described earlier, associations of templates and hooks can be mapped to meta-patterns. Also stated, meta-patterns capture the essence of design patterns by emphasizing the flexible aspects of these design elements. Taking these two facts into consideration, we have devised a reverse engineering process to detect design patterns.

According to Tourwé [30], tool support involving design patterns, first of all, requires a formal definition of those design patterns. A good abstraction is therefore mandatory and meta-patterns provide exactly the kind of abstraction required to define design patterns. This approach uses a formal representation of meta-patterns developed by Tourwé. In his work, a formal model for representing meta-patterns is presented. An example is presented below:

```
unificationFundamentalMP(H,H::Mt,H::Mh) with:
  participants:
    hookHierarchy(H)
    templatemethods(H::Mt)
    hookmethods(H::Mh)
  constraints:
    understandsMessage(root(H), H::Mt)
    definesMethod(root(H), H::Mh)
    understandsMessage(leafs(H), H::Mh)
    invokes(H::Mt, H::Mh, self)
```

The phased process unfolds as follows, each step leading into the next level of abstraction and presenting itself with deliverable results. Figure 2 depicts its successive steps and flow of information. It is intended to use the JUnit framework [5] as a test-case for results.

3.2 Source code parsing

At a first stage, the source code is parsed and all relevant information is gathered concerning candidate template and hook methods.

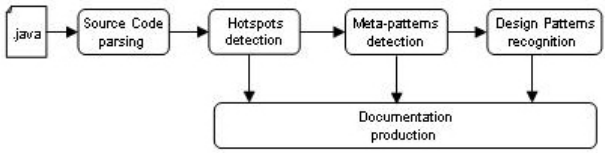

Figure 2: Reverse engineering process phases.

3.3 Hot spots detection

A second step groups the templates and hook methods into hot spots. At this level, information regarding the variation points of the framework is produced by a previously autonomous tool co-developed by the authors, called `HotSpotter` [12], which was adapted to integrate with JFREEDOM. A screenshot can be seen in Figure 3, where the HotSpotter has been run over the JUnit framework.

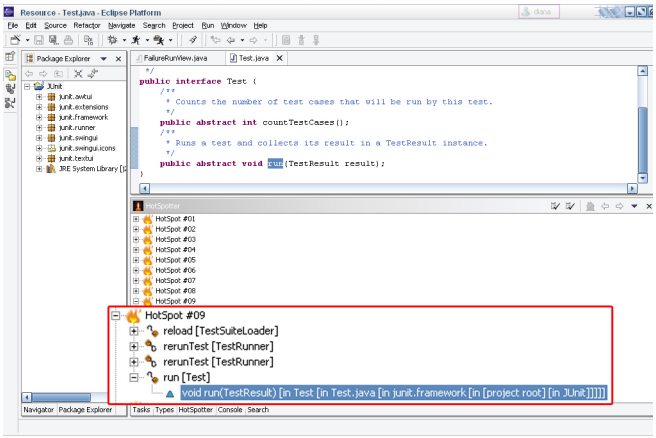

Figure 3: HotSpotter tool screenshot.

3.4 Meta-patterns detection

Using a formal definition of meta-patterns, template and hooks are matched into these representations. This step relies on a process of matching existing template/hook associations with the known meta-pattern representations, using an inference engine. Figure 4 shows an example of a known meta-pattern instantiation over JUnit.

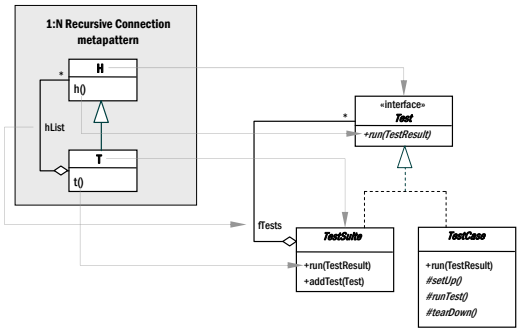

Figure 4: A meta-pattern instantiation in JUnit.

3.5 Pattern recognition

This step tries to match the meta-patterns with a formal design pattern representation based on meta-patterns previously stored in a knowledge base. The final result is expected to be similar to the one depicted in Figure 5 (extracted from the JUnit documentation).

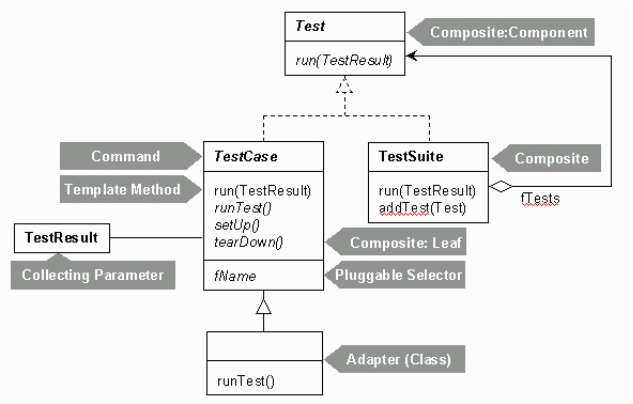

Figure 5: Design patterns of JUnit.

3.6 Documentation production

Each and every step will produce results, providing documentation over the identified framework variation points, possibly in several formats, such as UML [21], Javadoc [29], XSDoc [2], or SVG [11].

One of the main advantages of this approach is that, independently of the design patterns detection succeeding or not, the information produced at intermediate steps has added value and is "deliverable" as is, which includes increased information about the framework reusability.

Notice that the emphasis is on the reusability aspects of the framework. Even at the *Hot spots detection* step, there is useful and relevant information concerning the adaptable parts of the source code. Thus the process is, itself, flexible, leaving the user with the decision to proceed, or not, to the next level of abstraction.

By providing results in different formats, the approach widens its usage, allowing easier production of documentation at various levels.

4. JFREEDOM TOOL

The main goal of the work here presented is to develop a semi-automated tool to support this reverse engineering process. Existing available tools don't address reusability information in such a straightforward manner and are rather available for use.

The main concerns regarding this tool are:

- *flexible*, i.e., it should be pluggable into an integrated development environment.
- *interactive*, i.e., the user should be able to control its process and customize its behavior and preferences.
- able to *generate visual information*, whether graphics or text, to store the produced data.

As a result, JFREEDOM is being developed as an Eclipse plug-in, whose architecture is depicted in Figure 6.

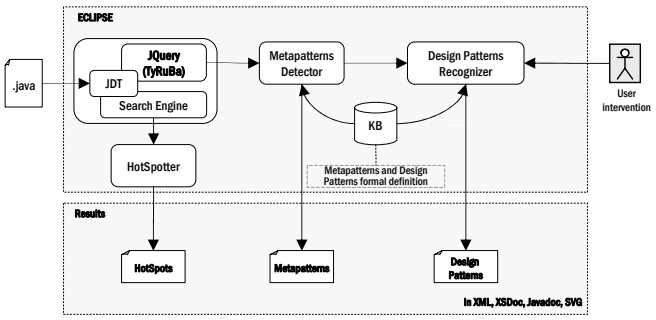

Figure 6: Architecture of JFREEDOM.

The tool is being developed in Java, using the TyRuBa inference engine [20] as a source code fact base. The TyRuBa engine was developed in conjunction with JQuery [19], a query-based code browser plug-in for Eclipse. It is defined as a set of TyRuBa predicates which operate on facts generated from the Eclipse JDT's abstract syntax tree.

At this stage of development, the second step of grouping templates and hooks into hot spots is done directly using the JDT Search Engine, by the `HotSpotter` module.

The subsequent modules, `Metapatterns detector` and `Design Patterns recognizer` use the inferred results from TyRuBa to produce their results. To deal with matching ambiguity, the user may modify the produced results, judging whether poorly matched patterns can be refined through the addition of extra design information of his/hers knowledge. Thus, the process can be continued with this new information in order to produce more accurate documentation.

5. RELATED WORK

The subject of reverse engineering design patterns has already undertaken several approaches in recent years. From automated detection tools to pattern extraction heuristics, most have a variable range of results regarding availability, flexibility, effectiveness and efficiency. A brief review over the existing approaches foung in the literature is presented below.

5.1 Pat

The *Pat*[9] system is a design recovery tool for C++. Extract design information from C++ header files and stores it in a repository. Patterns are defined as PROLOG rules and the design information is translated into facts. The actual matching work is done by a PROLOG engine. Its limitations lie on dealing with behavioral patterns, since too much semantic information is required.

5.2 KT

KT [6] is a tool that can reverse-engineer design diagrams from Smalltalk code and use this information to detect patterns. It supports both static and dynamic modeling information of design patterns. The methods used for the detection of patterns are hard-coded directly into the KT source, thus constraining its flexibility.

5.3 Seeman-Gudenberg

Seeman-Gudenberg [27] approach introduces several ideas on how to recover design information from Java source code. The method proceeds along successive steps, revealing different layers of abstraction. In a first step, a graph is generated after a source code parsing to collect information about inheritance, method calling and naming conventions. Next, this graph is transformed by graph grammar productions until pattern detection is finally applied. This approach relies on a Pipes & Filters architectural style.

5.4 SPOOL

SPOOL [25] (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project is a joint industry/university collaboration. The SPOOL reverse engineering environment has a three tier architecture: object-oriented database, repository schema and end-user tools. The lower tier provides physical storage of the reverse engineering model and design information. The middle tier contains the object-oriented schema of the reverse engineering model, comprising static structure and dynamic behavior. The upper-tier consists of end-user tools implementing domain specific functions such as source code capturing and visualization analysis.

5.5 Albin-Amiot & Guéhéneuc

Albin-Amiot & Guhneuc[3] approach bases itself on the definition of a meta-model to represent design patterns. This representation allows design patterns to be detected relying on their structural and behavioral aspects. The source code is parsed and an instantiation of the meta-model is produced accordingly. This instance is then matched against previously stored design pattern instantiations with the same meta-model.

5.6 JBOOTRET

JBOOTRET(Jade Bird Object-Oriented Reverse Engineering Tool) [17] uses a parser to extract the higher-level design information and conceptual model from system artifacts. The version for C++ consists of three major components: a data extractor, a knowledge manager and an information presenter. The approach intends to separate data extraction from information representation, thus preventing repeating the analysis process for each higher-level model extraction. This design gives a enhanced degree of flexibility, enabling an easy adaptation to other programming languages.

5.7 Heuzeroth-Holl-Hogstrom-Lowe

Heuzeroth-Holl-Hogstrom-Lowe [16] approach presents a way to automatically detect patterns by combining both static and dynamic analysis. The former restricts the code construction and the latter the runtime behavior. This analysis does not depend on coding or naming convention. A pattern instance is defined by a tuple of program elements such as classes, methods or attributes. These elements must conform to the rules of certain design patterns. A step-by-step process filters those elements more likely to be identifies as design patterns. Nevertheless, this automated process doesn't eliminate human intervention to ascertain the reliability of the results.

5.8 SPQR

SPQR [28] is a Pipes & Filters based approach where the source code is progressively filtered and converted into intermediate notations until it finally reaches a state suitable for Formal Proof assertions of design patterns. This process requires an extensive formalization of the design patterns made a priori, based on its structures and relationships between its components. Its representation relies on production rules. It is relevant to point out the reasonable high-abstraction level of this approach, similar to the meta-pattern level.

5.9 DPVK

DPVK [31] is a reverse engineering tool to detect design patterns in Eiffel source code. This approach relies on a phased process starting on a static behavior analysis of candidate design patterns and ending on a dynamic behavior analysis. It relies on information stored in a repository where the design patterns have previously been catalogued according to those two aspects of behavior. It presents itself as a plug-in for IBM's Eclipse development environment.

All these approaches deal with the aspect of flexibility and software reuse in a very shallow way, if at all. We are convinced that a design recovery approach relying on flexibility and reusability aspects provides a more clear insight over the usefullness of the framework purpose.

6. CONCLUSIONS

In order to efficiently understand a framework, one must be aware of its design, at different levels of abstraction. Most commonly, the accompanying design documentation of frameworks is poor or even inexistent, thus leading to a steep learning curve.

Uncovering the design of an existing framework is a hard and tiresome task, whereas an automated aiding tool comes in order. Several solutions exist to this problem, yet none deals with the aspect of reusability in a clear, straightforward manner, and are unable to provide useful intermediate results.

The approach proposed focuses purely on identifying the elements responsible for providing framework reusability and flexibility, at various levels of abstraction. Aimed at obtaining the same results as other existing approaches it provides however usable intermediate results (*hot-spots*, *template-hooks*, and *meta-patterns*), even if the final results (*design patterns*) aren't sufficiently accurate.

The proposed multi-phase process supports itself on the concept of meta-patterns, a meta-level representation of an abstract design pattern that describes the relationship between the elementary template and hook methods.

The automated tool aims at supporting the proposed approach, being flexible, interactive and visually delivering its results to the user. An Eclipse plug-in was decided to be the preferred way to satisfy such requirements.

It is the authors' conviction that this approach will make easier the reverse engineering of design patterns used in frameworks.

7. REFERENCES

[1] A. Aguiar. *A minimalist approach to framework documentation*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, September 2003.

[2] A. Aguiar, G. David, and M. Padilha. XSDoc: an Extensible Wiki-based Infrastructure for Framework Documentation. In E. Pimentel, N. R. Brisaboa, and J. Gómez, editors, *JISBD*, pages 11–24, 2003.

[3] H. Albin-Amiot and Y.-G. Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In B. Tekinerdogan, P. V. D. Broek, M. Saeki, P. Hruby, and G. Sunyé, editors, *proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[5] K. Beck and E. Gamma. Junit homepage, 1997. Available from http://www.junit.org.

[6] K. Brown. *Design reverse-engineering and automated design pattern detection in Smalltalk*. PhD thesis, North Carolina State University, 1996.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture — a System of Patterns*. John Wiley & Sons, 1996.

[8] G. Butler. A reuse case perspective on documenting frameworks. http://www.cs.concordia.ca/faculty/gregb, 1997.

[9] L. P. Christian Kramer. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, 1996.

[10] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999.

[11] J. Ferraiolo, F. Jun, and D. Jackson. Scalable vector graphics (SVG) 1.1 specification, w3c recommendation, January 2003. Available from http://www.w3.org/TR/SVG11.

[12] N. Flores, D. Soares, H. Ferreira, and M. Rodrigues. HotSpotter: : using JavaML to discover hot-spots. In *Proceedings of XATA 2005, XML: Aplicaes e Tecnologias Associadas*, February 2005.

[13] M. Fontoura, W. Pree, and B. Rumpe. The uml profile of framework architectures, 2000.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: abstraction and reuse of object-oriented design. In *Procedings of the ECOOP'93 Conference*. Springer-Verlag, 1993.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[16] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proceedings of 11th IEEE International Workshop on Program Comprehension*, pages 94–103, 2003.

[17] F. Y. Hong Mei, Tao Xie. Jbooret: an automated tool to recover oo design and source models. In *Proceedings of 25th Annual International Computer Software and Applications Conference*, pages 61–76, 2001.

[18] R. Johnson. Documenting frameworks using patterns. In A. Paepcke, editor, *OOPSLA'92 Conference Proceedings*, pages 63–76. ACM Press, Oct. 1992.

[19] Kris De Volder. JQuery Tool Home Page, 2004. http://www.cs.ubc.ca/labs/spl/projects/jquery/.

[20] Kris De Volder. TyRuBa Home Page, 2004. http://tyruba.sourceforge.net/.

[21] O. M. G. OMG. Unified modeling language, 2003. Available from http://www.omg.org/uml.

[22] W. Pree. *Object-oriented versus conventional construction of user interface prototyping tools*. PhD thesis, University of Linz, 1991.

[23] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley / ACM Press, 1995.

[24] W. Pree. Hot-spot-driven development. In *Building Application Frameworks — Object-Oriented Foundations of Framework Design*, pages 379–394. John Wiley & Sons, 1999.

[25] S. R.Keller, R.Schauer and P.Page. Pattern-based reverse engineering of design components. In *Proceedings of 21th Conference on Software Engineering*, pages 226–235, 1999.

[26] A.-M. Sassen and R. Marinescu. Metrics-based problem detection in object-oriented legacy systems using audit-reengineer. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 164–165, London, UK, 1999. Springer-Verlag.

[27] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of java software. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 10–16, New York, NY, USA, 1998. ACM Press.

[28] J. Smith and D. Stotts. Spqr: Flexible automated design pattern extraction from source code. In *In 18th IEEE International Conference on Automated Software Engineering*, 2003.

[29] Sun Microsystems. Javadoc Tool Home Page, 2003. http://java.sun.com/j2se/javadoc/.

[30] T. Tourwé. Automated support for framework-based software evolution, 2002.

[31] V. T. Wei Wang. Dpvk - an eclipse plug-in to detect design patterns in eiffeil systems. In *Department of Computer Science, York University, Toronto, Canada*, 2004.

[32] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. PH, 1990.