Program Comprehension and Design Pattern Recognition: An Experience Report

Francesca Arcelli and Claudia Raibulet
DISCo – Dipartimento di Informatica Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca, I-20126, Milan, Italy
{arcelli, raibulet}@disco.unimib.it

Abstract: The usefulness of techniques and tools supporting software evolution and maintenance is certainly of great relevance for program comprehension (i.e., "opportunistic code reading" [6]), reengineering, restructuring and re-documenting of large systems. Many tools have been proposed and developed supporting all the above tasks, but most of them do not work properly due to scalability problems. In this work we aim to describe our experience in understanding and extending a large system as Java PathFinder through its analysis with two reverse engineering tools: CodeCrawler and PTIDEJ.

1. Introduction

In the context of a Master level course at the University of Milano-Bicocca on "Software Evolution and Reverse Engineering", we had the possibility to experiment various tools of reverse engineering through the project examinations of the students. Our attention was concentrated in particular towards design pattern detection tools [2]. We used PTIDEJ (Pattern Trace Identification, Detection and Enhancement in Java) [1, 10] and FUJABA (From UML To Java And Back Again) [8]. An interesting approach is proposed by SPQR [13], which unfortunately is not available for testing. Another reverse engineering tool we intensively used for program comprehension is CodeCrawler [3]. We exploited appropriately the various polymetric views provided by this last tool.
We experimented the above tools on various open source projects such as: JUnit, JHotDraw, JEdit, Axis 1.0 Jigsaw, Java PathFinder (JPF), the MAIS project [7, 9], and on other projects under development at our university. To prove the different tools we decided to choose both very well designed systems (as JPF, JHotDraw) and projects still under development (as MAIS) which certainly need much more restructuring and redesigning efforts.
The objectives of this work are two:
- the first one is to describe our experience in using CodeCrawler and PTIDEJ to analyze JPF (see Section 3);
- the second one is to describe the benefits (*if any…*) we obtained through the above analysis during the project we started in collaboration with the NASA Automated Software Engineering Research Center [5], for the development of an Eclipse plug-in of JPF; this activity was carried out during a master thesis at our university [11, 12]. See Section 2 for a brief introduction on JPF and Section 4 for concluding remarks.

2. JPF and Eclipse

Java PathFinder (JPF) [5] is an explicit state software model checker for Java bytecode, developed by the Automated Software Engineering Group of NASA of the AMES Research Center. It is based on a virtual machine that executes software, theoretically, in all possible ways, checking for property violations as deadlocks, unhandled exceptions or user defined properties

along all potential execution paths/states. Its only limitation regards the impossibility of the JPF virtual machine to execute platform specific, native code. JPF offers many powerful extension, integration, and control mechanisms and, moreover, it is an open source project, so it is possible to integrate its verification engine in other applications by modifying the JPF core to adapt it to different requirements. Using Model Java Interface (MJI) it is possible to separate and communicate between state-tracked executions inside the Java Virtual Machine of the JPF.
Eclipse is probably the most important and used Java Integrated Developing Environment (IDE) and not only; it is a framework/environment that can be extended by a developer to integrate new functions, using the plug-in mechanism. Eclipse supplies a lot of *extension points*, which are used to interact with its environment and projects, or to add new graphical elements at its interface, supporting new functionalities. Using the JPF model checking engine in the Eclipse IDE it is possible to analyze software during its development phase through a friendly graphic user interface, ensuring developers that they are creating reliable software during the engineering and encoding phases.

3. Analyzing JPF through Reverse Engineering Tools

To develop our plug-in of JPF for Eclipse, we have first analyzed JPF exploiting two different tools: CodeCrawler [3, 6] and PTIDEJ [10]. The two tools are complementary in that CodeCrawler focuses on metrics and software visualization, while PTIDEJ on pattern detection and enhancement in Java.

3.1 Analysis through CodeCrawler

Generally, the analysis with CodeCrawler is performed at three different levels of granularity. The first level is considered course grained because it provides a view in terms of dimensions, complexity and structure of the entire system. The second level regards more specific systems aspects focusing on classes with particular dimensions and functions. The last one is considered fine grained because it allows the analysis of individual classes and of their implementation details.
In the context of our work, the course grained view has been particularly useful. It allowed us to have an overall view of the system without reading source code and, moreover, to identify immediately the classes belonging to the re-engineered simulation of the JVM. The system complexity view has changed significantly (see Figure 1 and Figure 2) because the new view consists only of the classes implementing the JPF business logic.
The second level of analysis helped us to understand that JPF concentrates its business logic into the root classes, which have the higher number of methods and the minimum number of descendants. The business logic of JPF resides in the following packages:

- open-jpf.jar, which implements the JPF model checking core;
- env-jpf.jar, which implements the execution environment of the JPF;
- env-jvm.jar, which implements the JPF simulation virtual machine.

In addition, through the *data storage class detection view* [3] we have identified the classes which implement the main functionalities of JPF and the classes we have modified or extended to implement our Eclipse plug-in.

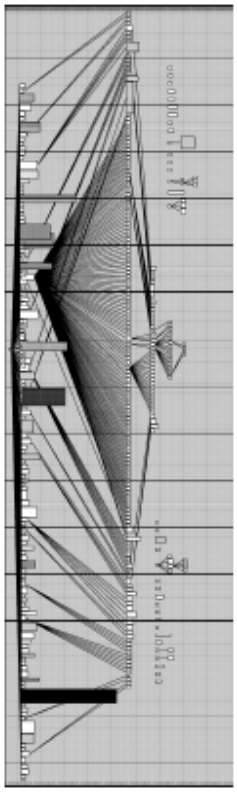

Figure 1. JPF Complexity View

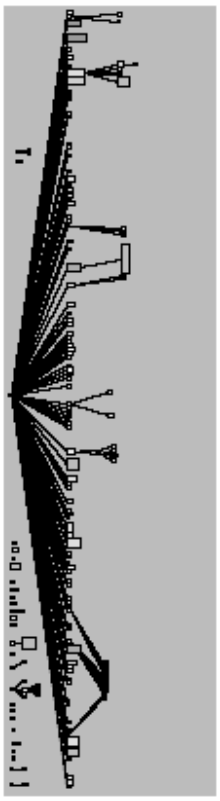

Figure 2. JPF Complexity View Without the JVM Package

To summarize, the analysis of JPF with CodeCrawler helped us to identify its main functional components: analysis, simulation engine, interfaces, simulation of the JVM, and so on, and the specific classes to interact with during the integration of JPF in the Eclipse environment.

3.2 Analysis through PTIDEJ

PTIDEJ is a design pattern recognition tool that enables users to understand better the rationale behind the design of the analyzed system and to evolve it by proposing improvements of the micro-architectures.

A limitation of PTIDEJ is that it works properly on a restrictive number of classes. Thus, we analyzed individually several packages of the JPF (i.e., ov.nasa.jpf.search, gov.nasa.jpf.symbolic.integer, gov.nasa.jpf.symbolic.string). We searched for the design patterns able to be recognized by the PTIDEJ: Facade, Mediator, Observer, Factory Method, Visitor, Chain of Responsibility, and Proxy. Facade and Proxy have been identified in the source code, but they are false positives.

PTIDEJ has correctly detected the Chain of Responsibility pattern in the package gov.nasa.jpf.symbolic.string. The highest percentage of 50 percent (see Figure 3) is coherent with the real micro-architecture presence. The classes involved are *Example* for the client role; it uses the function for string comparing of the class *StringComparator*. This class delegates the responsibility to the classes *Equal* or *EqualIgnoreCase*. The percentage is not equal to 100

because the class *StringComparator* not only delegates the task to other classes but executes other operations of a special class to give back the results attended by the client class *Example*.

Figure 3. Chain of Responsibility Detection Result

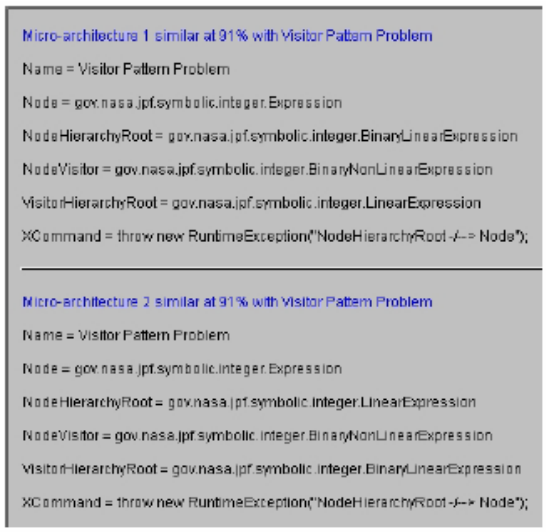

Figure 4. Visitor Detection Result

Another design pattern correctly recognized by PTIDEJ in the gov.nasa.jpf.symbolic.integer package is Visitor (see Figure 4). Although PTIDEJ does not detect the involved classes' roles with the same definition names in [4], it is possible, analyzing the UML package's structure and detecting interesting points in the source code, to say that a 91 detection percentage composing this micro-architecture is correct. The class *IntegerBinaryLinearExpression* is defined as the base of the visit hierarchy and if a visit of a binary or integer expression is performed, the access functionality is made using the correct visitor.

The Observer design pattern identified by PTIDEJ has an important role in JPF. One of the most important extension mechanisms of JPF are the Search/VMListerners, which provide a convenient way to extend JPF's internal state model by adding more complex properties checks, direct searches, or simply gathering statistics. These extensions are implemented exploiting the Observer pattern as shown in Figure 5.

The last useful result is related to the Mediator pattern detection in the package ov.nasa.jpf.symbolic.integer. This design pattern defines a class mediator to permit the communication of cooperating classes delegating to it the possibility to identify which classes and which communication modes to use. Moreover, it works to give new functionalities that are not present in the related classes. A 100 percentage is excessive in our example but it is possible, using the generated UML diagrams, to detect a structure which has the form and the functions of the design pattern definition. The *Expression* class has the mediator role for the classes defining logical and mathematical expressions.

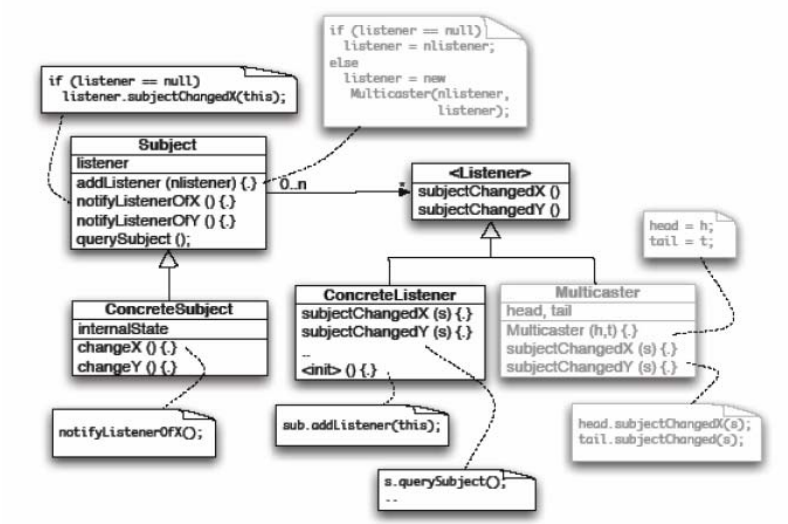

Figure 5. The Observer/Listener Design Pattern in JPF

4. Conclusions

The major advantage gained in using the above reverse engineering tools is related to the comprehension of JPF, to the correct understanding of how it really works. CodeCrawler has been particularly useful to have an overview of the entire application, to identify the main parts of the system and the functionalities they implement, as well as to identify the components that required further analysis. Moreover, through CodeCrawler we have identified the parts of JPF which are more or less independent to be given in input to PTIDEJ to recognize design patterns. Thus, the analysis using CodeCrwler was fundamental for the further analysis of the JPF. Furthermore, we proved that CodeCrawler can be successfully used to analyze large systems.

The design patterns detection helped us to understand better the design solutions chosen by the developers of JPF. PTIDEJ has detected several design patterns in JPF, the most useful one for our goal was the detection of Observer.

A great advantage of using these two tools derives from the resolution of several problems encountered during the development of the JFP plug-in for Eclipse. After the analysis of JPF, it was definitely easier to locate the sources of the problems in the code when trying to extend and execute JPF as a plug-in. We have encountered problems in launching and configuring JPF, when called by an external application. Analyzing the execution path of JPF, we have individuated the points that raised problems related to runtime class loading.

Due to the fact that also very well designed systems often suffer of the lack of very well documentation, we gained an additional advantage in using the above tools for recovering some software structures used during all the plug-in development and in particular in the integration phase.

We experiment also FUJABA for design pattern detection. The main problem of using FUJABA is that it detects a lot of false positives. Also PTIDEJ provides false positives, but this is less critical than in FUJABA. The only certain design pattern detected by FUJABA is Singleton, which denies the multiple creations of JPF instances by the same process.

Both the design pattern detection tools require a lot of system resources and a powerful computer to fulfill the search process quickly. The design pattern detection is performed through very intensive algorithms.

References

[1] H. Albin-Amiot, P. Cointe, Y. G. Guéhéneuc, and N. Jussien, "Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together", *Proceedings of the 16th International Conference on Automated Software Engineering*, San Diego, CA, USA, 2001, pp. 166-173.

[2] C. Chambers, B. Harrison, and J. Vlissides, "A Debate on Language and Tool Support for Design Patterns", *Proceeding of the 27th ACM SIGPLAN-SIGART Symposium on Principles of Programming Languages*, Boston, MA, USA, 2000, pp. 277-289.

[3] CodeCrawler - http://www.iam.unibe.ch/~scg/Research/CodeCrawler/index.html

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: elements of reusable object-oriented software*, Addison Wesley, Reading MA, USA, 1994.

[5] JPF - http://javapathfinder.sourceforge.net/

[6] M. Lanza, *Object-Oriented Reverse Engineering – Coarse-grained, Fine-grained, and Evolutionary Software Visualization*, PhD Thesis, University of Berna, 2003

[7] MAIS Project – http://www.mais-project.it

[8] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards Pattern-Based Design Recovery", *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, 2002, pp. 338-348.

[9] Pernici, B., (eds) *Mobile Information Systems - Infrastructure and Design for Adaptivity and Flexibility (the MAIS Approach)*, Springer-Verlag, 2006

[10] PTIDEJ - http://ptidej.iro.umontreal.ca/

[11] I. Rigo, F. Arcelli, C. Raibulet, L. Ubezio, "An Eclipse Plug-in for the Java PathFinder Runtime Verification System", accepted at the *30th Annual IEEE/NASA Software Engineering Workshop*, Columbia, Maryland, Metropolitan Washington DC, April, 24th – 28th, 2006

[12] I.Rigo. An Eclipse Plug-in for the Java Path Finder Runtime Verification System. Working on the NASA Model Checker. MS.Thesis, University of Milano Bicocca, Dec. 2005.

[13] J. McC. Smith, and D. Stotts, "SPQR: Flexible Automated Design Pattern Extraction From Source Code", *Proceedings of the 2003 IEEE International Conference on Automated Software Engineering*, Montreal QC, Canada, October, 2003, pp. 215-224