Ptidej: A Flexible Reverse Engineering Tool Suite

Demonstration – Approach

Yann-Gaël Guéhéneuc
Département d'informatique et de recherche opérationnelle
Université de Montréal – CP 6128 succ. Centre Ville
Montréal, Québec, Canada, H3C 3J7
guehene@iro.umontreal.ca

Abstract

The Ptidej *project started in 2001 to study code generation from and identification of design patterns. Since then, it has evolved into a complete reverse-engineering tool suite that includes several identification algorithms for idioms, micro-patterns, design patterns, and design defects. It is a flexible tool suite that attempts to ease as much as possible the development of new identification and analysis algorithms. In this demonstration, we first present the key features of the tool suite and several identification algorithms. We then discuss the architecture and design choices of the tool suite and lessons learned in developing the suite.*

1 Context

Maintenance of object oriented systems is a major concern both in industry and research. Despite the promises of object oriented (OO) programming, maintenance of OO systems amounts to 50 percent of the cost of programs and maintainers spend 75 percent of their time understanding code and documentation.

Maintainers must be aware of the design choices made during development to modify adequately a system. In particular, design choices include the structure of and relationships among classes.

However, design choices are often scattered in the code because, with available OO programming languages, they do not transcribe directly into code: developers usually must write many lines of code to implement their choices. Moreover, documentation, if any, is often obsolete.

Consequently, design choices are recovered manually, which is time and resource consuming. Fortunately, design choices are often implemented by following recurring patterns.

2 Problem

Maintainers need tools to recover design choices from code based on their originating patterns. These tools must be semi-automated considering the large size of current OO systems. There exist several types of patterns. Of interest in this demonstration are idioms, micro-patterns, design patterns, and design defects (code smells and antipatterns).

Idioms are low-level patterns specific to some programming languages and to the implementation of particular characteristics of classes or their relationships. They are intra-class patterns describing typical implementation of, for example, relationships, object containment, and collection traversal.

Micro-patterns have been recently introduced [6] as well-defined idioms pertaining to the design of classes in object-oriented programming.

Design patterns [5] are recurring inter-class patterns that define solutions to common design problems in the organisation of classes. They are defined in terms of classes and relationships using idioms.

Design defects are the "opposite" of design patterns. They describe "bad" solutions to recurring design problems. They can be low-level and are then called code smells [4] or higher-level, such as antipatterns [1].

3 Related Work

Several researchers have tackled the problem of identifying design choices and patterns. We cite here only three well-known tools for lack of space.

FAMOOS [2] is a complete and industrial-strength reverse-engineering environment based on Smalltalk. It includes a language-independent meta-model, FAMIX, and various identification algorithms. It is under constant development and evolution and has been used by many researchers across the world.

IntensiVE [9] is an environment that seamlessly integrates with a software development environment to support the definition, manipulation, and verification of intensionally defined sets of source code entities. It uses the SOUL [11] declarative meta-programming language, which has been used to identify design patterns.

Fujaba [10] provides an easy to extend UML, story-driven modelling, and graph transformation platform with the ability to add plug-ins. Experiments has been performed to add a pattern identification plug-in.

However, to the best of our knowledge, no existing tool offers algorithms to identify all the patterns of interest in this demonstration.

4 Solution

We demonstrate the PTIDEJ tool suite, a reverse-engineering environment designed to ease the development of pattern identification algorithms.

The main abilities of the tool suite are its flexibility and extensibility to allow various algorithms to live and interact harmoniously. As of 2007, the tool suite includes algorithms for idioms, micro-patterns, design patterns, and design defects.

The core of the tool suite is the PADL meta-model to represent OO systems and patterns with a unified language, which includes all constructs found in mainstream OO programming languages and is easily extensible. It includes parsers to built models of systems in AOL, C++, and Java.

Atop PADL are several levels of analyses, which depend on the data required by the analyses. The first level consists of PADL analyses performed directly on models of systems and patterns. It includes analyses to enrich a model with data on accessors and binary class relationships [8]. It also includes analyses to generate UML models where classes, methods, and fields are stereotyped according to the UML v1.5 specifications [7]. Finally, the identification of micro-patterns is defined as a PADL-level analysis.

The second level of analyses consists of UI-related analyses performed on models of systems and patterns to change their graphical representations. It includes analyses to highlight the differences between two models at the method- and class-levels. It also includes the identification of design patterns and defects.

Finally, the third level of analyses consists of UI extensions, allowing a richer interaction between the analyses and the user. It allow exporting models to external tools such as Dotty or Sugibib [3] as well as annotating models with user data.

The main novelties shown in this demonstration are the new user interface and its Sugiyama-based layout algorithm. We also show the model browser that allows analysing and displaying models of large systems.

5 Lesson Learned and Conclusion

The PTIDEJ tool suite attempts to ease the development of pattern identification algorithms; through its development we have learned several lessons.

First, flexibility is of major importance when developing a research tool. Future needs and research directions are unknown and thus it is important to divide the implementation into manageable, independent packages. The layered architecture is certainly the most suitable architecture.

Second, stability, in particular of the core level, is crucial to allow students and researchers to work with the tool suite. Indeed, students and researchers are "tough customers" who may be very demanding and loose interest quickly in front of buggy tools.

Finally, usability is a major concern. More and more, researchers are asked to create link with industry; involve B.Sc., M.Sc., and Ph.D. students; and share their tools. A usable UI helps in fostering a research community and disseminating ideas but is a real challenge to implement in a flexible way. For that reason, we choose to maintain our own tool suite rather than depend on third-party tools, such as Eclipse.

References

[1] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. M. III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1st edition, March 1998.

[2] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. In *proceedings of the 2nd UML conference*, pages 630–644. Springer-Verlag, October 1999.

[3] H. Eichelberger. *The syntax of UMLscript.* Department of Computer Science, University of Wuerzburg, October 2002.

[4] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1st edition, June 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1st edition, 1994.

[6] Y. Gil and I. Maman. Micro patterns in java code. In *proceedings of the 20th conference on Object-Oriented Programming Systems Languages and Applications*, pages 97–116. ACM Press, October 2005.

[7] Y.-G. Guéhéneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference.* IEEE Computer Society Press, November-December 2004.

[8] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proceedings of the 19th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[9] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views – a case study. In *Computer Languages, Systems, and Structures*, June 2006.

[10] J. Niere, J. P. Wadsack, and A. Zündorf. Recovering UML diagrams from Java code using patterns. In *proceedings of the 2nd workshop on Soft Computing Applied to Software Engineering*, pages 89–97. Springer-Verlag, February 2001.

[11] R. Wuyts, K. Mens, and T. D'Hondt. Explicit support for software development styles throughout the complete life cycle. Technical Report Vub-Prog-TR-99-07, Programming Technology Lab, Vrije Universiteit Brussel, April 1999.