

PTIDEJ and DECOR: Identification of Design Patterns and Design Defects

Naouel Moha and Yann-Gaël Guéhéneuc

Ptidej Team – GEODES Group
Département d'informatique et de recherche opérationnelle
Université de Montréal, Québec, Canada
{mohanaou, guehene}@iro.umontreal.ca

Abstract

The PTIDEJ project started in 2001 to study code generation from and identification of patterns. Since then, it has evolved into a complete reverse-engineering tool suite that includes several identification algorithms. It is a flexible tool suite that attempts to ease as much as possible the development of new identification and analysis algorithms. Recently, the module DECOR has been added to PTIDEJ and allows the detection of design defects, which are recurring design problems. In this demonstration, we particularly focus on the creation and use of identification algorithms for design patterns and defects.

Category and Subject Descriptors D.2.7 Software engineering; Distribution, maintenance, and enhancement; Restructuring, reverse engineering, and reengineering

General Terms Algorithms, design, languages.

Keywords Design patterns, design defects, antipatterns, meta-modelling, detection, Java.

1. Introduction

Quality is an important goal in development process of all applications, from games to life support systems. Quality is usually assessed and improved during formal technical reviews, primarily to detect errors and defects early, before they are passed on to another software activity or released to the customer [15]. Similarly, the application of design patterns allows software engineers to design and implement systems of high stability and quality [9].

Design patterns [5] describe good solutions to common and recurring problems in program design. Design defects are the “opposite” of design patterns. They describe solutions to recurring problems that generate negative conse-

quences on the quality of object-oriented systems [1]. They can be low-level and are then called code smells [3] or higher-level, such as antipatterns [1].

Maintainers must be aware of the design choices or defects made during development to modify adequately and improve a system. However, their manual identification in large programs is a time-, resource-consuming, and error-prone activity. Maintainers need tools to recover design choices or defects from code based on their originating patterns. These tools must be semi-automated considering the large size of current OO systems.

2. Related Work

Several researchers have tackled the problem of identifying on one hand design patterns and the other hand defects. We cite here only well-known tools for lack of space.

IntensivE [12] is an environment that seamlessly integrates with a development environment to support the definition, manipulation, and verification of intensionally defined sets of source code entities. It uses the SOUL [16] declarative meta-programming language, which has been used to identify design patterns. Fujaba [13] provides an easy to extend UML, story-driven modelling, and graph transformation platform with the ability to add plug-ins. Experiments have been performed to add a pattern identification plug-in.

Marinescu [10] presented a metric-based approach to detect code smells with detection strategies, implemented in a tool called iPLASMA. However, metrics cannot express important structural relationships and properties, while in DECOR, detection rules can also be specified using structural and lexical properties and structural relationships. Moreover, the iPLASMA tool was only evaluated in terms of precision; no recall was computed. Tools such as CHECKSTYLE [2], FXCOP [4], and PMD [14] detect problems related to coding standards, bugs patterns or unused code. They focus on implementation problems but do not address higher-level design defects such as antipatterns.

However, no existing tool offers algorithms to identify several different kinds of patterns, including design defects.

3. PTIDEJ and DECOR

We demonstrate the PTIDEJ¹ 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 algorithms to live and interact harmoniously. As of 2007, the tool suite includes algorithms for idioms, micro-patterns, design patterns, and design defects. Idioms are low-level patterns specific to some programming languages and to implementation of particular characteristics of classes or their relationships. Micro-patterns have been recently introduced [6] as well-defined idioms pertaining to the design of classes in object-oriented programming. 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 build models of systems in AOL, C++, and Java.

Our tool suite for the identification of design patterns revolves around a constraint solver to identify micro-architectures similar to design motifs in programs, which are the solutions advocated in the design patterns. In previous work [7, 8], we described design motifs as constraint systems: Each role is represented as a variable and relationships among roles are represented as constraints among variables. Variables had identical domains: All the classes in a program in which to identify design motifs. For example, the identification of micro-architectures similar to the Composite design motif, shown in Figure 1, in a given program translates to the constraint system :

Variables:	Constraints:
client	association(client, component)
component	inheritance(component, composite)
composite	inheritance(component, leaf)
leaf	composition(composite, component)

where the four variables `client`, `component`, `composite`, and `leaf` have identical domains and the four constraints represent the association, inheritance, and composition relationships suggested by the Composite design motif.

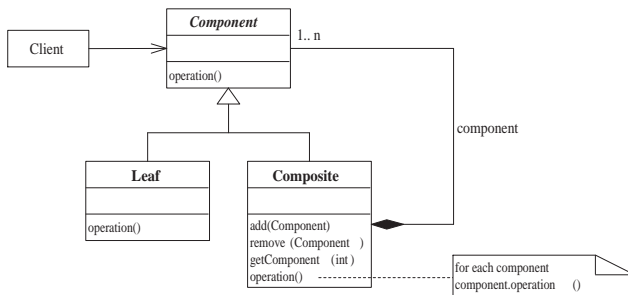


Figure 1. The Composite design motif [5]

¹ PTIDEJ stands for *Pattern Trace Identification, Detection, and Enhancement in Java*.

To identify design defects, in our approach DECOR², we use a domain-specific language to specify and generate automatically design defect detection algorithms using templates [11]. A domain-specific language offers greater flexibility than ad hoc algorithms because the domain experts, software engineers, can specify and modify manually the detection rules using high-level abstractions, taking into account the context, environment, and characteristics of the analysed systems. We show that the language is straightforward to understand and handle and allows the automated generation of detection algorithms. The main novelties shown in this demonstration are the rule editor to create, edit, and compile new detection algorithms for design defects and the use of these algorithms on several programs. We also demonstrate the precision and recall of our algorithms.

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*, 1998.
- [2] CheckStyle, <http://checkstyle.sourceforge.net>, 2004.
- [3] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1999.
- [4] FXCop, <http://www.gotdotnet.com/team/fxcop/>, 2006.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1994.
- [6] Y. Gil and I. Maman, “Micro patterns in java code,” in *Proceedings of the 20th OOPSLA Conference*, 2005.
- [7] Y.-G. Guéhéneuc and H. Albin-Amiot, “Recovering Binary Class Relationships: Putting Icing on the UML,” in *Proceedings of the 19th OOPSLA Conference*, 2004.
- [8] Y.-G. Guéhéneuc, “A systematic study of UML class diagram constituents for their abstract and precise recovery,” in *Proceedings of the 11th APSEC Conference*, 2004.
- [9] P. Kuchana, *Software Architecture Design Patterns in Java*, 2004.
- [10] R. Marinescu, *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws* in *Proceedings of the 20th International Conference on Software Maintenance*, 2004.
- [11] N. Moha, Y.-G. Guéhéneuc, and P. Leduc, “Automatic generation of detection algorithms for design defects,” in *Proceedings of the 21st ASE Conference*, 2006.
- [12] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts, “Co-evolving code and design with intensional views – a case study,” *Computer Languages, Systems, and Structures*, 2006.
- [13] J. Niere, J. P. Wadsack, and A. Zündorf, “Recovering UML diagrams from Java code using patterns,” in *Proceedings of the 2nd workshop on SCASE*, 2001.
- [14] PMD, <http://pmd.sourceforge.net/>, 2002.
- [15] R.S. Pressman, *Software Engineering – A Practitioner’s Approach*, McGraw-Hill Higher Education, 2001.
- [16] R. Wuyts, K. Mens, and T. D’Hondt, “Explicit support for software development styles throughout the complete life cycle,” Vrije Universiteit Brussel, Tech. Rep. Vub-Prog-TR-99-07, 1999.

² DECOR stands for (*Defect dEtECTION for CORrection*). Correction is out of the scope of this demonstration.