This paper has been accepted at the ASE 2002 workshop on Declarative Meta-Programming.

Three Musketeers to the Rescue

Meta-modelling, Logic Programming, and Explanation-based Constraint Programming for Pattern Description and Detection

Yann-Gaël Guéhéneuc[⋆]

École des Mines de Nantes
4, rue Alfred Kastler – BP 20823
44307 Nantes Cedex 3
France
`guehene@emn.fr`

1 Introduction

Software maintenance is a costly and tedious phase in the software development process [37]. During this phase, a maintainer needs both to understand and to modify a program source code. Therefore, she needs a representation of the program that accurately reflects its structure and its behavior. Then, she must find those places in the program that require modifications. Finally, she must perform changes that improve the program behavior and that do not introduce further defects.

In our research work, we focus on the maintainer's first and second tasks: The obtention of an accurate representation of the program structure and behavior, and the detection of places to improve. We propose a set of software engineering tools for the representation of Java programs (structural and dynamic information), and for the (semi-) automated detection of design patterns and design defects. Design patterns and design defects are related: We assume that a group of classes whose micro-architecture is similar (but not identical) to a design pattern corresponds to a possible design defect [16]. Either the pattern is distorted because it has been clumsily implemented, or the pattern is distorted because it does not fit in the architecture: In each case, the maintainer may want to analyze further the highlighted micro-architecture.

We concentrate on programs developed using object-oriented programming languages, because we are interested in detecting (object-oriented) design patterns and design defects. However, we believe we could apply our approach to any programming language (functional, procedural...) given suitable elements to describe patterns and programs written with this programming language.

⋆ This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

We develop:

- PatternsBox, a tool to describe design patterns and design defects, to apply design patterns, and to detect complete forms of design patterns [1].
- Caffeine, a tool for the dynamic analysis of Java programs. We develop a library to analyze binary class relationships [18].
- Ptidej Solver, an explanation-based constraint solver [24] to detect design patterns and design defects [19].
- Ptidej, a tool to visualize program architecture and behavioral information, to generate the problems related to the detections of design patterns and of design defects, and to display results of the detections.

These tools use different declarative meta-programming paradigms. According to the Merriam-Webster's (2002), a paradigm is "*a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated*". We interpret this definition and we use the term declarative meta-programming to denote programs (and languages) which subjects of computation are programs or program artifacts:

- In PatternsBox, we describe patterns using a meta-model. The descriptions represent the structure and the behavior of the *Solution* elements of the patterns. Descriptions are first-class entities, which we can manipulate (to generate source code...) and reason about (to compare with other descriptions...).
- In Caffeine, we analyze a program execution using queries. We express the queries in term of a trace model and of an execution model of the program execution, using logic programming. The Caffeine system runs as a co-routine of the program to analyze, which emits events that abstract its runtime behavior. The Caffeine system receives, analyzes, and drives the program execution according to the queries.
- In Ptidej Solver, we use explanation-based constraint programming [24] to solve problems representing the detection of design patterns and design defects in a program architecture. A problem decomposes in a constraint system, generated from a pattern, and in a domain, representing the program architecture. The constraint system and the problem domain are generated automatically from the pattern and the program architecture, both expressed using the PatternsBox meta-model.
- In Ptidej, we display program architectures in terms of the PatternsBox meta-model and of behavioral information, obtained using Caffeine. We display classes, inheritance and implementation relationships, and binary class relationships, such as the association, the aggregation, and the composition relationships [17]. We visualize the results of the detection computed by the Ptidej Solver.

In this position paper, we present and motivate the use of the three distinct declarative meta-programming paradigms: Meta-modelling, logic programming, and explanation-based constraint programming. We also describe a succinct scenario highlighting the use of our tools.

2 Pattern Description

Patterns have been widely accepted by software practitioners. They cover all phases of the software development process: Requirements [20]; Analysis [12]; Architecture [6]; Design [15]; Implementation (idioms) [7]; Defects [5]; Refactoring [13]; Testing [14].

Requirements, analysis, and architectural patterns are high-level, informal, and general-purpose patterns; Whereas design, defect, and implementation patterns are tightly coupled with the software development process and implementation. Thus, it is desirable to formalize these patterns [2, 30, 32]. Formalization may support the reification, the instantiation, the application, the detection, and the comparison of patterns. Several techniques exist to formalize design, defect, and implementation patterns:

- Logic programming [27, 38].
- Logic-based notations [11].
- Meta-modelling [2, 25, 28, 34]
- Program generation [36].
- Program transformations [35].
- UML-based and associated notations (*eg.*, OCL) [15, 31, 34].
- State charts [15].
- Protocols and finite-state machines.

The choice of a particular technique depends on the intended use of the formalization. In PatternsBox, we want to reify design patterns and design defects as first-class entities from declarative descriptions of the patterns structure and behavior. Then, we want to reason and to interact with the patterns: To instantiate them; To display them (either as source code, as constraint systems, or as constraint domains...); And, to recognize them in source code. Therefore, we choose the meta-modelling technique to represent patterns.

Other authors use different techniques to formalize patterns. However, these techniques have shortcomings *w.r.t.* our intended use of the formalization. In a system based on logic programming, a set of predicates describes a pattern, but the pattern is not reified within the system: We cannot reason about it or interact with it from *within* the system. For example, code generation takes place in a separate module, which input is the set of predicates.

3 Dynamic Information

Program analysis is an important issue in object-oriented software engineering. Program analysis serves different purposes, such as extracting object-model from source code [22], understanding dependencies among classes and their instances [17], or verifying, refuting, simulating, and checking properties [21]

Program analysis may be performed either statically or dynamically. On the one hand, some information may be costly, complex, or even impossible to extract from static program-analysis. On the other hand, the results of dynamic analyses are valid only for the set of considered executions.

In Caffeine, we want to perform dynamic analyses of Java programs to extract information on the dependencies among classes and their instances. We model the execution of a program as a trace, which is a history of execution events. Execution events abstract the execution of the program as Prolog predicates, for example `fieldModification(...)`, `finalizerEntry(...)`, or `programEnd(...)`. We request the next available events through a Prolog engine, which runs as a co-routine of the program being analyzed. The Prolog engine drives the execution of the program under analysis using the Java platform debug architecture API [33]. Prolog already showed its adequacy to query traces in different works [9, 10].

For example, the following Prolog predicates count the number of times method `startTest` executes:

```
query(N, M) :-
    nextEvent(
        [generateMethodEntryEvent],
        E),
    E = methodEntry(_, startTest, _, _, _),
    N1 is N + 1,
    query(N1, M).
query(N, N).

main(N, M) :- query(N, M).
main(N, N).
```

First, we order Caffeine to obtain the next `methodEntry` event from the program execution, using the `nextEvent` predicate. Second, we filter out method entries corresponding to the `startTest` method, using the `=` predicate. Third, we increment a counter and recursively call the query. The use of logic programming allows a powerful and quite natural expression of queries on the trace of the program execution. In particular, we develop a set of predicates to verify properties on binary class relationships [17].

Other techniques to reason about program executions include universally quantified predicates [26], regular expressions, or temporal logic [8]. However, for our purpose, each one of these techniques has drawbacks. Universally quantified predicates are more declarative than Prolog queries, but they only deal with state information. Regular expressions are simpler and more efficient but with less power of expression. Temporal logic eases expressing temporal relationships (such as precedence) but is less expressive than logic programming, which offers a unification mechanism and high-level pattern matching capabilities.

4 Pattern Detection

The automated detection of patterns in source code is a difficult task and is subject of many works, such as [3, 4, 23, 29]. When patterns are used in a program architecture, there is no real links between the patterns (their actors and the relationships among them) and the source code (the classes and the relationships among them).

In PTIDEJ SOLVER, we use the information collected from PATTERNSBOX and CAFFEINE to represent a program architecture. From this information and given a pattern described using the PATTERNSBOX meta-model, we automatically generate a constraint problem where the constraints and the variables correspond to the pattern to detect, and where the domain correspond to the classes and the relationships among the classes of the program architecture.

We extend and use a constraint solver with explanations [19, 24], PALM, to obtain automatically all the complete and distorted solutions to the constraint problem, even if the constraint problem is over-constrained or if there exists no solution with all the constraints. Distorted solutions are solutions to a subset of the given constraints and thus represent possible design defects *w.r.t.* the design pattern being detected.

For example, the PALM code excerpt below instructs the constraint solver to compute solutions to the problem of *Good Inheritance*. The *Good Inheritance* pattern states that an entity `Super-entity` (class or interface) must not know[1] about any other entity `Sub-entity` that extends (or implements) it.

```
let pb := makePtidejProblem("Good Inheritance", length(listOfEntities), 90),
    superEntity := makePtidejIntVar(pb, "Super-entity", 1, length(listOfEntities)),
    subEntity   := makePtidejIntVar(pb, "Sub-entity", 1, length(listOfEntities)) in (
    post(pb, makeStrictInheritancePathConstraint(
             subEntity,
             superEntity),
             100),
    post(pb, makeIgnoranceConstraint(
             superEntity,
             subEntity),
             50))
```

First, we declare a new problem, which domain is the number of entities in the program architecture, `length(listOfEntities)`, and which maximum level of constraint relaxation is 90. Second, we declare the variables of the problem: Two variables `superEntity` and `subEntity`, which values range from 1 to `length(listOfEntities)`. Finally, we post two constraints:

- The first constraint, `StrictInheritancePath`, states that the two variables must instantiate such that the entity in variable `subEntity` extends (or implements) the entity in variable `superEntity`.
- The second constraint, `Ignorance`, states that the two variables must instantiate such that the entity in variable `superEntity` does not know about the entity in variable `superEntity`.

We assign a weight of 50 to the `Ignorance` constraint to allow the PTIDEJ SOLVER to remove this constraint when it fails to find more solutions. The solutions found without the `Ignorance` constraint corresponds to distorted solutions to the *Good Inheritance* pattern: These solutions are possible design defects.

[1] The precise definition of the knowledge relationship is out of the scope of this article, the interested reader may refer to work [17].

The use of explanation-based constraint programming makes it easy to express complex problems in terms of the solutions we want rather than how to compute the solutions. Also, it eases the explanation of distorted solutions and thus the detection of possible design defects.

Explanation-based constraint programming is more powerful than other approaches such as fuzzy logic [23] or logic programming [38]. Fuzzy logic proved its usefulness for detecting defects in class declarations, but generic fuzzy reasoning nets seem difficult to construct and they require fine tuning. Logic programming only helps in detecting classes whose relationships are described by the logical rules: It does not *directly* help in detecting distorted solutions.

5 Example

We now present a scenario highlighting the use and integration of our different tools: Caffeine, PatternsBox, Ptidej, and Ptidej Solver[2].

A maintainer desires to understand better the architecture of JUnit v3.7 and to find possible design defects. We assume that the maintainer starts from scratch, with no pattern described yet. We also assume that she has a good knowledge of the design patterns in [15].

She turns to Ptidej and she loads JUnit v3.7 to display its architecture. She quickly browses the program architecture and notices the `TestResult` class that possesses a container-aggregation relationship with the `TestListener` interface[3] (both classes from package `jtu.framework`). She wonders if this container-aggregation could, in facts, be a composition-container relationship.

She turns to Caffeine and writes a simple program that uses Caffeine to analyze the relationship between classes `TestResult` and `TestListener` with a specific Prolog query [17]. She runs her program with the `MoneyTest` test class, provided with JUnit v3.7, as input and she obtains the confirmation that the relationship between classes `TestResult` and `TestListener` is indeed a composition-container relationship.

She goes back to Ptidej and she loads the result of the dynamic analysis. The model of the architecture changes to reflect the behavioral information. She recognizes that such a container-composition relationship between two classes is the sign of a (possible) implementation of the Composite design pattern: She decides to verify this possibility. First, she builds a meta-entity describing the *Solution* element of the Composite design pattern, using constituents of the PatternsBox meta-model. Second, she uses PatternsBox to interact with the Composite meta-entity. She instantiates the meta-entity into an abstract model. She could parameterize the abstract model to fine-tune the model but she decides to go on with the abstract model as it is. She chooses the PaLM custom-constraint builder from the set of available builders and save the constraint system associated with the abstract model of the Composite design pattern to disk. Third,

[2] For the sake of place, we only summarize their use.

[3] For a discussion on binary class relationships, the interested reader may turn to [17].

she generates the domain corresponding to the architecture of JUnit v3.7 and calls the Ptidej Solver. The Ptidej Solver computes the set of complete and distorted solutions.

Finally, the maintainer loads the solutions to the constraint problem and browses the two different distorted solutions found by the constraint solvers. The solutions are, respectively, close at 60% and 1% to the micro-architecture advocated by the Composite abstract model. She must now further investigate the micro-architectures highlighted by the constraint results and decide whether or not these micro-architectures represents a Composite design pattern and whether or not modifications are required.

6 Conclusion

Declarative meta-programming is at the core of our software engineering tools. We conjointly use meta-modelling, logic-base programming, and explanation-based constraint programming to solve very practical software engineering problems: The declaration of patterns, the representation of programs, and the detection of patterns in the source code of programs.

Acknowledgements

The author deeply thank Hervé Albin-Amiot for the PatternsBox tool and its meta-model, and his kind support and invaluable advices.

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proceedings of ASE*, pages 166–173. IEEE Computer Society Press, November 2001.

[2] H. Albin-Amiot and Y.-G. Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of the ECOOP Workshop on Automating Object-Oriented Software Development Methods*. University of Twente, The Netherlands, October 2001. TR-CTIT-01-35.

[3] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. *Proceedings of the 6 th Workshop on Program Comprehension*, pages 153–160, 1998.

[4] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

[5] 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, Inc., 1998.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, Inc., 1996.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[8] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report KSU CIS Technical Report 2001-04, Kansas State University, 2001. Submitted for journal publication.

[9] M. Ducassé. Coca: A debugger for C based on fine grained control flow and data events. In *Proceedings of ICSE*, pages 504–513. ACM Press, May 1999.

[10] M. Ducassé. OPIUM: An extendable trace analyser for Prolog. In *The Journal of Logic Programming, Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, volume 41, pages 177–223. Elsevier – North Holland, November 1999.

[11] A. H. Eden, A. Yehudai, and J. Y. Gil. Precise specification and automatic application of design patterns. In *Proceedings of ASE*, pages 143–152. IEEE Computer Society Press, November 1997.

[12] M. Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley Object Technology Series, 1996.

[13] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] E. Gamma and K. Beck. JUnit. Available at: `http://www.junit.org/`, 2002. Available at: `http://www.junit.org/`.

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

[16] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of TOOLS USA*, pages 296–305. IEEE Computer Society Press, July 2001.

[17] Y.-G. Guéhéneuc, H. Albin-Amiot, R. Douence, and P. Cointe. Bridging the gap between modeling and programming languages. Technical Report 02/09/INFO, École des Mines de Nantes, July 2002.

[18] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In *Proceedings of ASE*. IEEE Computer Society Press, September 2002.

[19] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *Proceedings of the IJCAI Workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[20] A. Isazadeh, G. H. MacEwen, and A. J. Malton. Behavioral patterns for software requirement engineering. In *CD-Rom on CASCON*. Centre for Advanced Studies of IBM Toronto Laboratory and the Institute for Information Technology of the National Research Council of Canada, November 1995.

[21] D. Jackson and M. C. Rinard. Software analysis: A roadmap. In *Proceedings of ICSE, Future of Software Engineering Track*, pages 133–145. ACM Press, June 2000.

[22] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *Proceedings of ICSE*, pages 194–202. ACM Press, May 1999.

[23] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. *Proceedings of the European Software Engineering Conference*, pages 193–210, 1997.

[24] N. Jussien. e-Constraints: Explanation-based constraint programming. In *CP'01 Workshop on User-Interaction in Constraint Satisfaction*, December 2001.

[25] T. Kobayashi. Object-oriented modeling of software patterns and support tool. In *Proceedings of the ECOOP Workshop on Automating Object-Oriented Software Development Methods*. University of Twente, The Netherlands, October 2001. TR-CTIT-01-35.

[26] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In *Proceedings of ECOOP*, pages 135–160. Springer-Verlag, June 1999.

[27] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 2002.

[28] B.-U. Pagel and M. Winter. Towards pattern-based tools. *Proceedings of EuropLop*, 1996.

[29] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–883, December 1998.

[30] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 2000.

[31] P. Rapicault and M. Fornarino. Instanciation et vérification de patterns de conception : Un méta-protocole. *Proceedings of LMO, in French*, pages 43–58, 2000.

[32] J. Soukup. *Implementing Patterns*, chapter 20. Addison-Wesley, 1995.

[33] Sun Microsystems, Inc. Java platform debug architecture, 2002. Available at: `http://java.sun.com/ products/jpda/`.

[34] G. Sunyé. *Mise En Oeuvre de Patterns de Conception : Un Outil*. PhD thesis, Université de Paris 6 – LIP6, July 1999.

[35] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98, Vancouver, Canada*, pages 56–60, October 1998.

[36] K. D. Volder. Implementing design patterns as declarative code generators. Submitted at ECOOP 2001, 2001.

[37] S. G. Woods, A. E. Quilici, and Q. Yang. *Constraint-Based Design Recovery for Software Reengineering – Theory and Experiments*. Kluwer Academic Publishers, Kluwer Academic Publishers Group, Distribution Center, Post Office Box 322, 3300 AH Dordrecht, The Netherlands, 1998.

[38] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.