Ptidej: Promoting Patterns with Patterns

Yann-Gaël Guéhéneuc
GEODES - Group of Open and Distributed
Systems, Experimental Software Engineering
Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
guehene@iro.umontreal.ca

Abstract

We introduce the Ptidej *project and its tool suite to evaluate and to enhance software quality by promoting patterns. First, we summarise the components of the tool suite and describe its implementation in Java, which uses several architectural, design, and language patterns. Then, we take position on issues related to pattern claims, choices, uses, and limits from our experience with pattern definition, formalisation, use for reverse-engineering and for implementation.*

1 Introduction

The Ptidej[1] project (*Pattern Trace Identification, Detection, and Enhancement in Java*) aims at developing a tool suite to evaluate and to enhance the quality of object-oriented programs, promoting the use of patterns, at language-, design-, or architectural-level [4].

So far, the tool suite allows, through its user-interface, to create a model of a program from its source code, to identify micro-architectures similar to a design pattern, and to call various generators, analyses, and external tools on the program model. Figure 4, on the last page, summarises the functioning of the tool suite.

The Ptidej project decomposes in:

- A meta-model, PADL (*Pattern and Abstract-level Description Language*), to describe the structure of motifs—the "Solution" parts in pattern definitions—and of object-oriented programs.
- A library of design motifs from design patterns [6], including Chain of Responsibility, Composite, Observer, Visitor...
- Several parsers to build models of programs from different source code representations, including AOL [1], C++ files and Java class files.
- A library of software metrics, POM (*Primitives, Operators, Metrics*), to compute well-known metrics on models of programs, such as Chidamber and Kemerer's metrics [3].
- A library of generators and analyses to be applied on models of programs and of motifs.
- An explanation-based constraint solver [10], Ptidej Solver, to identify micro-architectures similar to motif models in program models.
- A dynamic analyser for Java, Caffeine, based on a Prolog engine and the Java debug interface to define relationships among classes precisely.
- A library of graphic widgets, Ptidej UI, to display models of motifs, of programs, and dynamic data from Caffeine.
- Several user-interfaces (see Figure 3 page 8 for an example) to access the functionalities provided by the Ptidej tool suite:
  - Parse and create models of programs.
  - Enhance models of programs with dynamic data from program executions.
  - Visualise created models.
  - Identify micro-architectures similar to a design motif model in a program model.
  - Visualise the identified micro-architectures.
  - Call generators (*i.e.*, Java generator), analyses (*i.e.*, a systematic UML-like analysis), and external tools (*i.e.*, Dotty) on models.

In Section 2, we summarise the implementation of the Ptidej tool suite, focussing on the choice and use of patterns. Then, in Section 3, we discuss pattern claims, choices, uses, and limits. Finally, in Section 4, we conclude with our positions on patterns and some open questions we would like to discuss.

[1] Ptidej stands for "breakfast" (in French argot) and is pronounced "tE-dAzh (as in *Pte*rodactyl and *Déj*à vu).

2 Implementation of Ptidej

In this section, we present the implementation of the Ptidej tool suite, which uses architectural, design, and language patterns. We focus on the main classes and packages and on the patterns without specifying every implementation details.

In particular, we do not detail two important tools of the Ptidej tool suite: Caffeine and Ptidej Solver because these tools have been described elsewhere (see [8] and [9], respectively) and because these tools do not participate *directly* to the architecture of the Ptidej tool suite and thus to the patterns used.

2.1 Architecture

We chose a layered architecture to build the Ptidej tool suite. A layered architecture allows us to build generators, analyses, and user-interfaces on a stable and well-tested meta-model.

Thus, we decompose the tool suite in two parts: One part related to the PADL meta-model and one part related to the Ptidej UI graphic framework.

The Ptidej tools suite is implemented 100% in Java, using the Eclipse development platform [13]. It decomposes in about 30 projects, 190 packages, 1,150 classes for 74,000 lines of code.

2.2 Design

The PADL Meta-Model. Figure 1 next page shows a UML-like class diagram representing the architectural layers corresponding to the PADL meta-model, their main packages and classes, and the design patterns used in the design.

The diagram decomposes in three horizontal parts representing three different layers of services: First, CPL (*Common Ptidej Library*); Then, PADL; Finally, PADL ClassFile Creator, PADL AOL Creator, POM, and PADL Analyses.

The first layer, CPL, declares utility classes and libraries used across the whole Ptidej tool suite, such as CFParse [7] and Javassist [2]. In particular, this layer declares the `ClassLoader` and `SubtypeLoader` classes, which we use to handle files and directories uniformly. This layer also declares the `util.awt` package, which classes are used to handle user interaction in various places. The dependency on Java AWT does not prohibit independence from the graphic library because Java AWT is always present in our target runtime settings (J2SE or J2EE, not J2ME).

The second layer, PADL, declares the meta-model to describe models of programs and of motifs. The meta-model defines several constituents (for example, `IIdiomLevelModel`, `IMethod`) that are interfaces which implementations are combined to describe structural models of programs and of patterns (and subsets of their behaviours).

The constituents of the meta-model are loaded in the tool suite dynamically: The implementations of the `IFileRepository` interface serves to access the constituents uniformly whether they are available in the form of class-files, of a JAR file...

The `padl.kernel` and `padl.kernel.impl` packages declares respectively the types of the constituents (as Java interfaces, hence the 'I' in front of each name) and their implementations. We use the Abstract Factory design pattern to manage the concrete instantiation of the constituents, the concrete factory, class `Factory`, follows the Singleton design pattern. We use the Builder design pattern to let the parsers choose the constituents to instantiate, through the `Builder` class. We use the Visitor design pattern to offer a standard mean to iterate over a model or a subset of a model, the `padl.visitor` package provides default visitors. The `padl.pattern` and `padl.pattern.repository` packages define several prototypal models of well-known design motifs, which we can clone and parameterise, using the Prototype design pattern.

The third layer contains several separate projects:

- Parsers for Java class-files and AOL files (PADL Java and AOL Creator). These parsers are independent of the meta-model and new parsers for other programming languages can be added seamlessly using the Builder design pattern.
- A metric computation framework (POM), in which we use the Singleton design pattern. POM decomposes in a set of primitives defined in terms of the meta-model constituents. These primitives are combined using set operators to define metrics.
- A repository of analyses based on the meta-model, in which we use a simpler version of the Command design pattern. An analyse is invoked on a model of a program or of a pattern and returns a (potentially modified) model when the analysis is done. Reflection is used by the repository to build the list of available analyses dynamically.

The Ptidej UI Graphic Framework. Figure 2 page 5 shows a UML-like class diagram representing the architectural layers corresponding to the Ptidej UI graphic framework and Ptidej user-interfaces, their main packages and classes, and the design patterns used in the design.

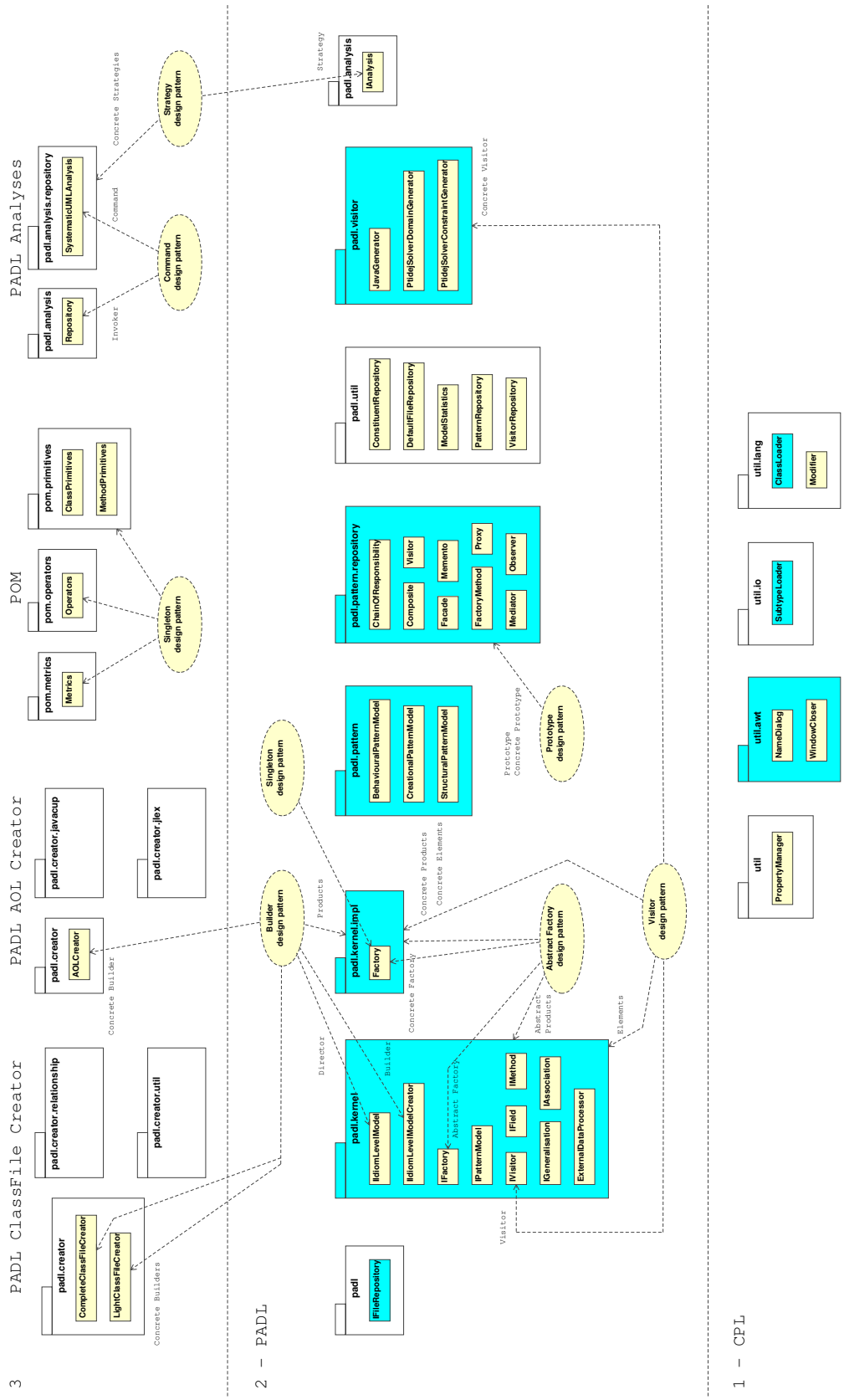

Figure 1. The PADL meta-model layers

As with the PADL meta-model, the diagram decomposes in horizontal parts representing four different layers of services: First, PTIDEJ UI; Then, PTIDEJ UI PRIMITIVES for AWT and SWT; PTIDEJ UI VIEWER and its EXTENSIONS; Finally, PTIDEJ UI VIEWER PLUGIN, APPLET, and STANDALONE.

The first layer, PTIDEJ UI, declares the graphic widgets used to display models of programs and of motifs. The `Canvas` class, along with the `ptidej.ui.event` and `ptidej.ui.canvas.event` packages, handles graphic representations of models and manages events using the Observer design pattern. The `Canvas` class accepts any implementations of the `IDrawable` and `ISelectable` (if appropriate) interfaces, which the `ptidej.ui.kernel` package provides to represent constituents of the meta-model graphically. Classes in the `ptidej.ui.kernel` package are independent of any graphic library through the graphic primitives from the `ptidej.ui.primitive` package, which are instantiated indirectly using the Abstract Factory design pattern. We use the Builder design pattern to build a graphic representation of a model with the `Builder` class. We use the Strategy design pattern to offer several layout algorithms through implementations of the `IGraphLayout` interface.

The second layer declares two implementations of the graphic primitives in package `ptidej.ui.primitive`, for JAVA AWT and for SWT (the graphic library particular to the ECLIPSE platform [13]). We use the Abstract Factory design pattern to manage instantiations.

The third layer declares user-interface utility classes, with PTIDEJ UI VIEWER, and offers an extension mechanism to graphic representations of models of programs and of motifs, in PTIDEJ UI VIEWER EXTENSIONS. The `ptidej.viewer` package declares state holders for models and their graphic representations, including micro-architectures similar to patterns. The `ptidej.viewer.event` package allows broadcasting changes to the data by applying the Observer design pattern. We use the Strategy design pattern to build the `IExtension` and `Repository` classes and the `ptidej.viewer.extension.repository` package to offer extensions to graphic representations.

The fourth layer declares three user-interfaces, respectively, as an ECLIPSE plug-in, as a Java stand-alone program (see Figure 3 page 8), and as a Java applet. User-interfaces are responsible for providing a graphic-dependent canvas that forwards appropriate messages to the `Canvas` class and for using the correct graphic primitives concrete factory. (The applet user-interface reuses the canvas of the stand-alone user-interface). They can implement functionalities to build and to manage models of programs and of motifs, their graphic representations, model analyses, and user-interface extensions.

2.3 Idioms

Idioms are language-level patterns. They are commonly used in many programming languages to propagate "good" programming styles.

We follow idioms advocated by the Java community, see for example `java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`. In addition, we promote:

- The use of the English language in class/method/field declarations as well as in documentation.
- A common formatting style, using the ECLIPSE Java source code formatter.
- A consistent commenting of the code.
- Consistent naming conventions for parameters, instance/local variables, class variables, and constants.
- A minimality in the imports: No "star" imports.
- The declaration of all instance variables as private.
- The systematic qualification of method calls and field accesses, using `this` for instance attributes or the class-name for class attributes.
- The systematic use of the `final` keyword for class/field/parameter/local variable declarations.
- The systematic use of the Iterator design pattern rather than `for` loop to iterate over lists.
- The systematic use of the `finally` clause when handling exceptions.
- The use of stricter compilation rules, in particular for unused private methods/fields and unused local variables

3 Discussions on Patterns

We now attempt to provide answers to questions regarding patterns from our experience in using patterns to build a tool suite to detect patterns.

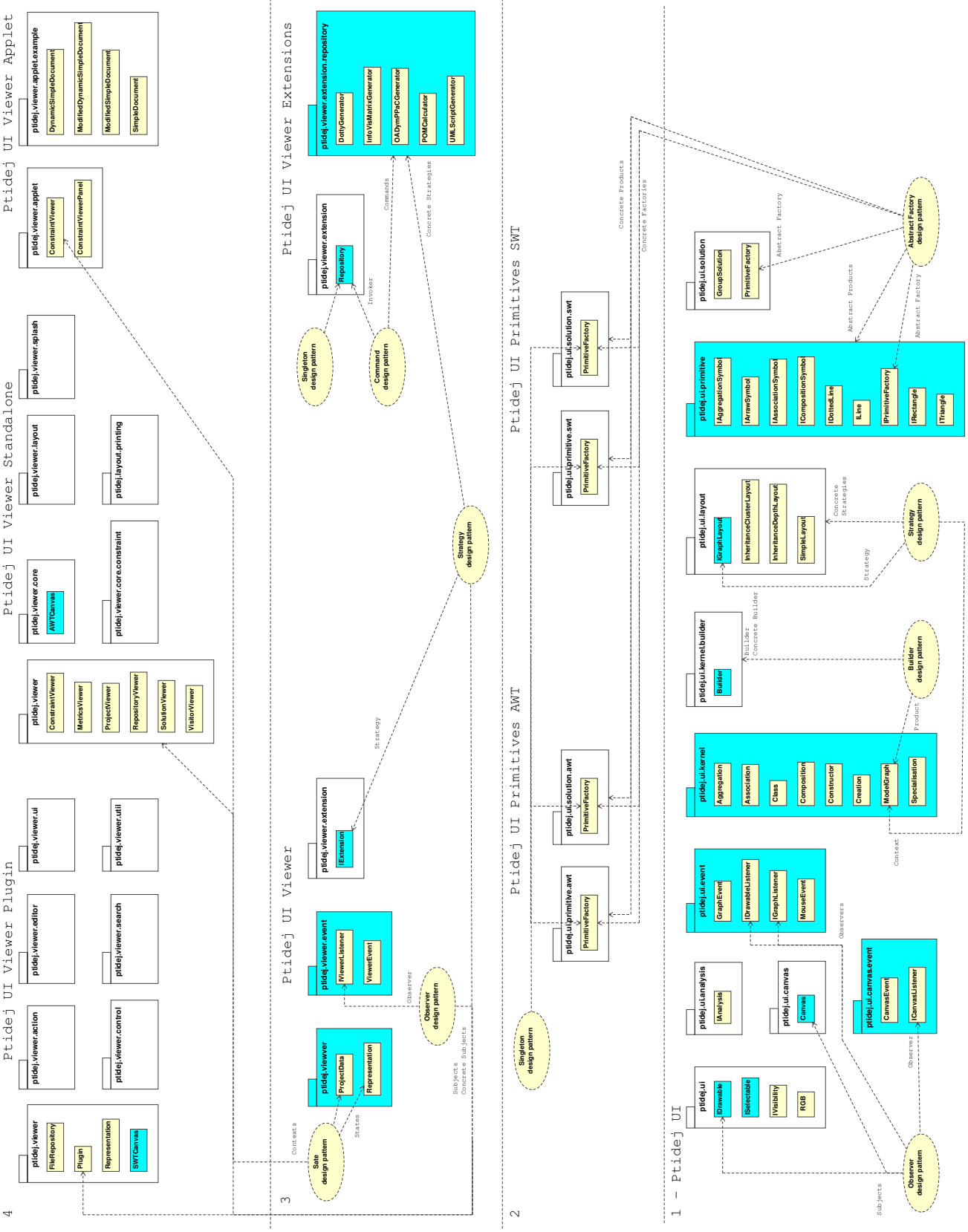

Figure 2. The Ptidej UI graphic framework and user-interfaces

Are the various claims related to building any system from patterns reasonable? In our understanding, the claims regarding patterns and software development concerns the flexibility, the reusability, and the understandability of the program implementations from the *developers*' point of view.

We agree that using patterns, either at architectural, design, or language levels, makes a program implementation more flexible and reusable (ease of adding and–or modifying behaviour of the program).

However, we are uncertain about the increase or decrease of understandability of the program implementation. On the one hand, with a knowledge on patterns and on the patterns used in a particular implementation, a developer can grasp quickly the collaborations between various parts of the program. On the other hand, (1) developers need to be really comfortable with many patterns before they can take advantage of their knowledge and (2) documentation does not often describe explicitly used patterns and their automated identification is a difficult problem.

Thus, we believe that *most* claims about patterns are reasonable but others are still overrated because of the lack in teaching and in tooling.

What do we mean when we say "systems of patterns"? We think of a system of patterns as a set of patterns that can collaborate together without too many contradicting intents and implementations.

Thus, we believe that subsets of the design patterns from the Gang of Four [6] form systems of patterns because they can be used in combination without conflicts, for example, Abstract Factory, Builder, and Singleton. However, all the design patterns do not form a system of patterns because of conflicting intents.

The difficulty to define systems of pattern rest principally on the lack of formalisation of the patterns, in particular of the "Intent", "Motivation", and–or "Consequences" parts of pattern definitions.

Thus, we believe that the pattern community must strive to formalise patterns better (not only their motifs, *i.e.*, their "Solution" parts) to provide unambiguous means to relate patterns with one another.

What are the various claims related to patterns composition; Are they true? In our understanding, the claims regarding pattern composition concerns the possibility to compose patterns and the composition flexibility, reusability, and understandability.

We believe that these claims relate to systems of patterns. Indeed, within a system of patterns, patterns can be composed while retaining their qualities, while among different systems of patterns, some patterns may conflict and thus reduce flexibility, reusability, and understandability.

We performed an experience with bachelor students to identify design patterns in several program implementations. We obtained best results when the students actually looked for one design pattern and then used the "Pattern Map" in the GoF's book to look for other related design patterns, which comfort our idea that composition relates to systems of patterns.

However, the concrete demonstration of pattern composition, apart from actual experiences, requires an extensive formalisation of patterns.

If someone would like to build a system from patterns, how do you select patterns? We strongly believe in the piecemeal growth of program implementations [5]. A program implementation evolves in time with maintenance and the needs for patterns may appear and disappear, and so do patterns.

In the development of the PTIDEJ tool suite, we begun with a very simple and straightforward implementation of the meta-model and, as work progressed, we used various patterns to increase flexibility, reusability, and understandability.

What kind of patterns should one select to build a system from patterns? As we advocate piecemeal growth of program implementation, we believe that developers should use any appropriate patterns for the task at hand.

The difficulty of such an approach resides in the possible lack of knowledge on useful patterns and, thus, in the need of pattern education.

Is there a guideline for the selection process? We found useful to compare possibly competing patterns when selecting patterns. In particular, we use our knowledge of patterns and a tool, DP TUTOR [12], to assess the adequation of one pattern or another in a specific context.

Are there any existing techniques for integrating patterns into traditional development cycles? We use and we promote two techniques to integrate pattern in development cycle: Reverse-engineering and refactorings.

Our tool suite assume that a better understanding of a program implementation at the design and architectural levels helps in applying patterns and, thus, in bringing flexibility, reusability, and understanding.

We concur with Kerievsky [11] that refactorings are a mean to integrate patterns seamlessly with software development cycles.

4 Position and Conclusion

Our position is that, as a community, we must promote patterns in our software development and, as importantly, in our teaching. We must work on pattern formalisation and develop tools to identify and to apply patterns. Thus, we can promote flexibility, reusability, and understandability—important qualities indeed—in program implementations.

We would like to discuss the interests, opportunities, and techniques of identifying and of detecting patterns (semi-)automatically. We would like also to discuss pattern education.

Also, we are concerned with the terminology surrounding patterns: "leitmotiv", "motifs", "instance", "occurrence" are but a few of the terms used to talk about patterns, most of them with no clear definition. We believe that the community must clearly defines these terms if we want to be able to compare related work on a common basis.

Acknowledgement

The author thank gratefully Hervé Albin-Amiot for his early work on the meta-model and all the students who participated so far in the development of the PTIDEJ tool suite: Salime Bensemmane, Ward Flores, Denise Gbetibouo, Jean-Yves Guyomarc'h, Duc-Loc Huyhn, Khashayar Khosravi, Lulzim Laloshi, Naouel Moha, Emmanuelle Orcel, Samah Rached, Sébastien Robidoux, Driton Salihu, Iyadh Sidhom, Fayçal Skhiri, Yves Bia Toe, and Farouk Zaidi.

References

[1] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In Scott Tilley and Giuseppe Visaggio, editors, *proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] Shigeru Chiba. Javassist – A reflection-based programming wizard for Java. In Jean-Charles Fabre and Shigeru Chiba, editors, *proceedings of the OOPSLA workshop on Reflective Programming in C++ and Java*. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management, December 1993.

[4] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In Laurie Dillon and Walter Tichy, editors, *proceedings of the* 25^{th} *International Conference on Software Engineering*, pages 149–159. ACM Press, May 2003.

[5] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1^{st} edition, April 1996.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[7] Matt Greenwood. *CFParse Distribution*. IBM AlphaWorks, September 2000.

[8] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the* 17^{th} *conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.

[9] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *proceedings of the* 1^{st} *IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[10] Narendra Jussien. e-Constraints: Explanation-based constraint programming. In Barry O'Sullivan and Eugene Freuder, editors, 1^{st} *CP workshop on User-Interaction in Constraint Satisfaction*, December 2001.

[11] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 1^{st} edition, August 2004.

[12] Olivier Motelet. An intelligent tutoring system to help OO system designers using design patterns. Master's thesis, Vrije Universitët, 1999.

[13] Object Technology International, Inc. / IBM. Eclipse platform – A universal tool platform, July 2001.

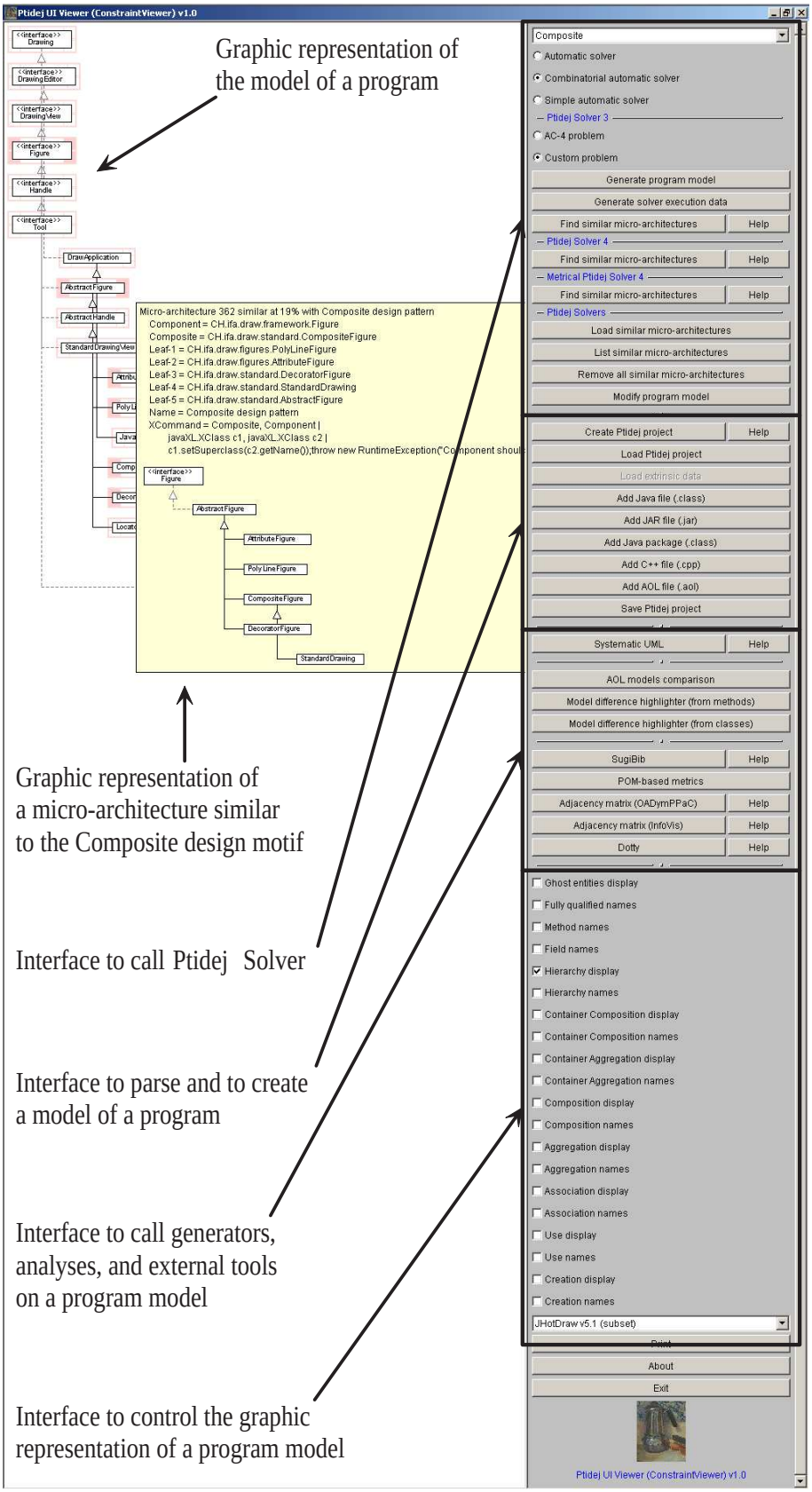

Figure 3. Ptidej interface

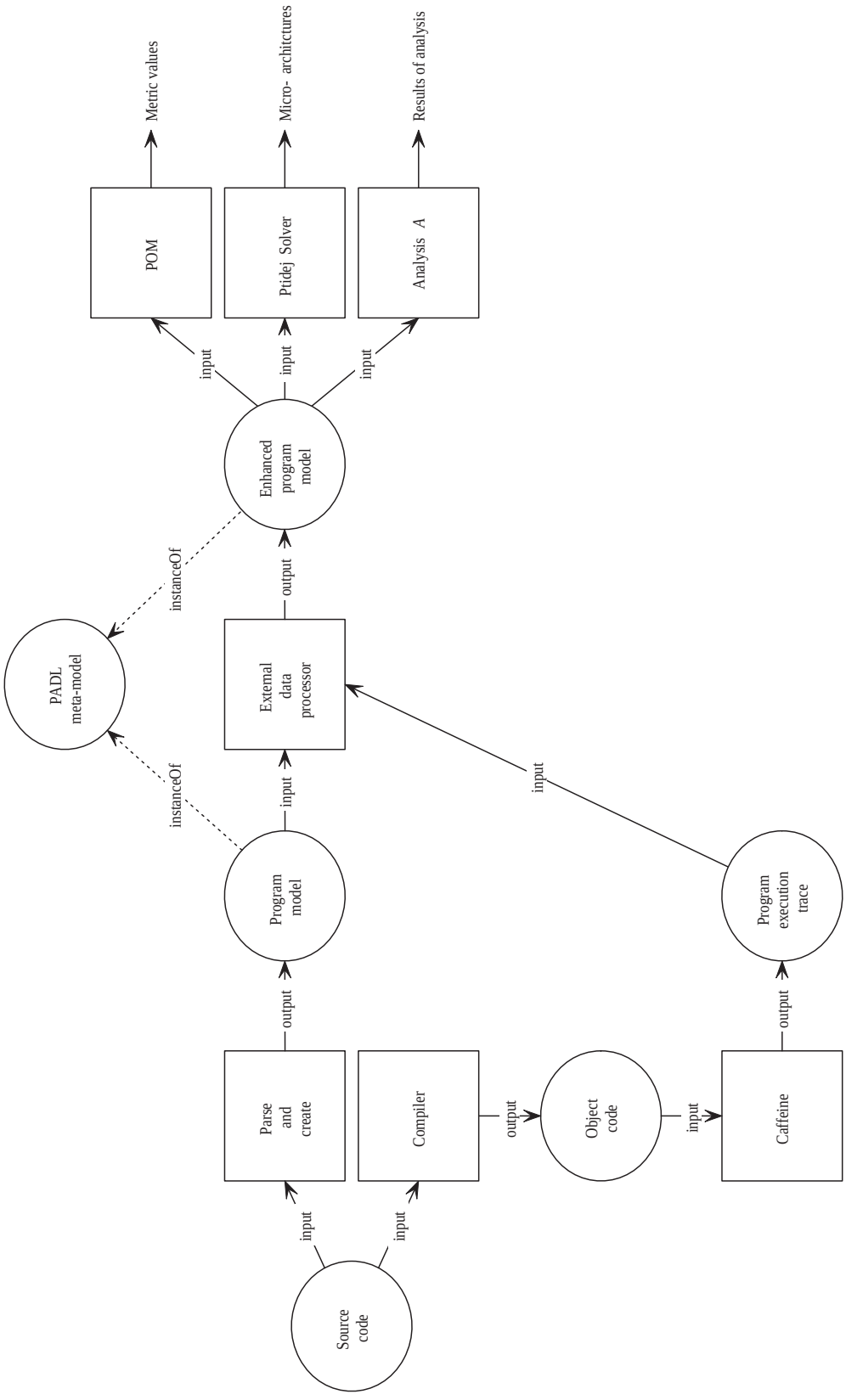

Figure 4. Use of the Ptidej tool suite