

Patterns of the Agenda

Amnon H. Eden¹

Amiram Yehudai¹

Abstract: *Most of the study of patterns has been restricted to the composition of "new" patterns. These authors, however, believe that the investigation of design patterns is ripe for an endeavor at an underlying formal conceptual framework. In this article we address future directions in the investigation of design patterns and stress the significance of precise specifications. We also propose our own approach at precise specification of design patterns.*

Keywords: Tool support for design patterns, formal methods, metaprogramming

1. Introduction

Patterns and design patterns have been attracting great interest within the OOP community and the general software industry. Their research is, however, yet at its cradle, and effort is mostly constrained to the definition (or discovery) of new software patterns (see pattern mining [GoV 96]). It is the humble opinion of these authors that much benefit can be gained from the study of other questions that would typically be classified as second-order or reflexive reasoning about patterns.

It is often claimed that generalizations about design patterns cannot be validated without having a solid body of patterns; there seems to be no dispute in this point. Nonetheless, we believe that with the accumulated publications of design pattern papers [GoF 96; Coplien & Schmidt 95; Vlissides, Coplien & Kerth 96; GoV 96] there is no longer a justification to defer treating design patterns with what are widespread scientific techniques. We conclude that the investigation of design patterns is apt for a fundamental, comparative, and formal analysis.

2. Open Questions

This section discusses topics in the understanding of design patterns that are yet to be studied. These questions will serve us later in measuring the adequacy and usefulness of a theory of design patterns. The questions we present are the following:

1. Is it possible to formalize the specification of a design pattern? In particular, the problem described? Its solution?
2. What are the constraints that rule the application of each design pattern? The outcome of such application? The composition of different patterns?
3. Among the activities related to design patterns are:
 - The **implementation** (application) of a design pattern

¹ The Department of Computer Science, School of Mathematics, Tel Aviv University, Tel Aviv, Israel. Email: {eden,amiram}@math.tau.ac.il

- The **recognition** of a design pattern, i.e., tracking the “existence” of a known pattern in a certain program
- The **discovery** of a new design pattern, i.e., tracking a repeated formation of behavior and/or structure in a certain program

How, if possible, can these activities be automated? For instance, can they be supported by a CASE tool?

4. How can design patterns be effectively indexed or classified?

We believe that precise specification of design patterns is a prerequisite to their analysis and the automation of their implementation.

3. Elements Of The Specification Of Design Patterns

The purpose of this section is to distinguish the role of the generic solution proposed by each pattern from the problems it attempts to solve.

Problems vs. solutions

Two basic elements partake in the specification of design patterns: *problem* and *solution*. The problem (usually accompanied by descriptions of *forces*, *context*, or *applicability*) part is usually defined in vague terms, demonstrated using one or two examples. This is followed by (code and diagram) examples that serve as an approximation to the complete problem domain. The solution is mainly described using natural language specification that attempts to circumscribe its applicability.

To characterize the recurring motifs in structure and behavior specified by a particular design pattern, we designate this construct as *lattice*². Principally, a *lattice* is the “scaffolding” of class fragments and specialized relations among them as indicated by a solution proposed in a design pattern³. The term *lattice* distinguishes the generic solution from the rest of the pattern.

Many justly object to referring exclusively to the pattern’s lattice; it is claimed that doing so allegedly abolishes the very purpose of patterns: to disseminate good design and experience gained in solving recurring problems. Indeed, this purpose (the dissemination of solutions) is best served by the contemporary nature of design patterns, as the overwhelming success of the [GoF 96] book and other patterns’ literature indicates. Nonetheless, it is the lattice that most effectively sets off one design pattern from another. To stress our point we quote the following two definitions for design patterns:

“Description of communicating objects and classes that are customized to solve a general design in a particular context [GoF 95]”

² No connection to the respective mathematical term.

³ A precise definition would be possible when a formal foundation is provided, such as the one proposed in section 4.

“Design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context [Coplien & Schmidt 95]”

Both definitions emphasize the solution element of the pattern form, and appear as synonyms to *lattice* as defined above.

A design pattern is best identified with the respective lattice. Note, for instance, that the set of problems a particular design pattern address may change without affecting the identity of the pattern. For example, pattern “sequels” (such as [Kim & Benner 95; Rohnert 96]) elaborate only on the description of the original problem, leaving the description of the solution intact⁴. Also observe that the lattice cannot change without radically effecting the pattern. Finally, a design pattern never proposes more than a single lattice, while it may address various problems.

We believe the reasons listed above justify a formal treatment of the *lattice* concept in separation from its respective design pattern.

4. The Metaprogramming Approach

According to the metaprogramming approach, a design pattern’s lattice is represented by a sequence of operations performed over elements of a program, such as classes and relations, as prescribed by this pattern. In other words, the description of a lattice is transformed to the application algorithm, by which it is represented.

Two programming languages partake in the metaprogramming scheme: The *metalanguage* is the language by which the manipulation of programs is phrased; for this purpose we used Smalltalk in our prototype, as an expressive and flexible language with a powerful class library. The *object language* is the language of programs to which lattices are introduced. We used Eiffel for our prototype as a convenient, static object oriented programming language (OOPL).

The metaprogramming approach incorporates also a class library, designated “the internal representation”, or the abstract syntax, instances of which classes represent programs in the object language. The combination of the abstract syntax with the Smalltalk language is an example to a *pattern specification language* (PSL), forming an API to the user who wishes to specify how a lattice is implemented.

A lattice is represented as a routine in PSL with arguments, often existing constructs in a program that are manipulated and adjusted to incorporate the lattice. The routine’s body specifies the necessary modifications to the program’s and possibly introduces new constructs to it. The result of the enactment of the routine is the program modified to incorporate an implementation of the respective lattice.

We implemented a prototype for a tool that supports the specification and application of design patterns in PSL, the ‘pattern’s wizard.’ The Smalltalk-80 environment,

⁴ At least in the sense listed above, i.e., recurring motifs in structure and behavior.

combined with the abstract syntax, provides a convenient, dynamic metaprogramming apparatus. The wizard's role is also to support the application of lattices in the object language.

Definition: A *trick* is an operator defined over elements of the abstract syntax; in other words, it is a unit of modification of a program. A *trick* is defined as a PSL routine, the body of which may comprise other tricks of the same abstraction level or lower. Tricks may have more than one version, each of which has a distinct *synopsis*.

The specification of each trick defines a respective lattice, as the abstraction of the result of the application of this trick. For example, the lattices prescribed by the *Visitor* and the *Decorator* patterns [GoF 95] can be induced by the enactment of suitable tricks. In [Eden, Gil & Yehudai 97] we demonstrate the specification of the tricks that ultimately induce the *Visitor* pattern and additional design patterns.

Our Ultimate intent is to define a hierarchy of *tricks* of different abstraction levels. At the first level we introduce *micro-patterns*, tricks of small scale that are defined as routines in PSL, each of which must be decidable. We strive to define a finite, small set of micro-patterns that is sufficiently expressive to identify the lattices of distinguished design patterns at the next levels, such as the *Visitor*, the *Abstract Factory*, and the *Observer*..

Recognizing Occurrences of a Lattice

The decidability of tricks is a condition that is necessary to allow their recognition. It remains to be solved whether, given a “fundamental” set of “suitable” (i.e., decidable) micro-patterns, it is possible to find the sequence of specific trick enactments (or the their manual equivalents) that lead to a given program. A possible problem is that this search process may result in more than one solution, each of which is a valid sequence of enactments of tricks of the given set. We, however, consider this particular case an example to an insight gained by the recognition process.

5. Conclusions

We claimed that a suitable, small set of micro-patterns, can account for lattices of most design patterns. We have proved that the specification of tricks allows the implementation of various lattices in source code. It remains to be proved whether the use of tricks forms a foundation for the definition of lattices and for their recognition. The results so far indicate that this method of formalization of patterns has the potential to address what we posed at the beginning of this article as open questions that the research of patterns has yet to resolve.

Acknowledgments

We are grateful to Dr. Gil for his useful remarks and to Prof. Hirshfeld for his contribution. This research was enabled in part by the German Israeli Fund (GIF).

References

- Beck, K. and R. Johnson (1994). *Patterns Generate Architecture*. European Conference on Object Oriented Programming. Berlin: Springer-Verlag.
- GoV: Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal (1996). *Pattern-Oriented Software Architecture-A System of Patterns*. Wiley & Sons.
- Coplien, J. O. and D. C. Schmidt (1995), eds. *Pattern Languages of Program Design*. Addison-Wesley.
- Eden A. H., J. Gil and A. Yehudai (1997). *Precise Specification and Automatic Application of Design Patterns*. Automatic Software Engineering - ASE'97.
Also available at:
http://www.math.tau.ac.il/~eden/precise_specification_and_automatic_application_of_design_patterns.{ps,Z,rtf}.zip
- Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley.
- GoF: Gamma E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
- Johnson, R. & W. Cunningham (1995). *Introduction*, in: [Vlissides, Coplien & Kerth 96].
- Kim, Jung J and Kevin M. Benner (1995). *Implementation Patterns for the Observer Pattern*, in: [Vlissides, Coplien & Kerth 95].
- Rohnert, H. (1996). *The Proxy Design Pattern Revisited*, in: [Vlissides, Coplien & Kerth 96].
- Vlissides, J. M., J. O. Coplien, and N. L. Kerth (1996), eds. *Pattern Languages of Program Design 2*. Addison-Wesley.