A Formal Language for Design Patterns

(extended abstract)

Amnon H. Eden[1]
Joseph (Yossi) Gil[2]
Amiram Yehudai[1]

Abstract

This article sets forth a rigorous approach to the concept of a 'design pattern language.' We present a method of using a programming language to manipulate constructs of ordinary OOP language (such as classes and relations). We show that the specification of many design patterns can be phrased as algorithms in this metaprogramming environment. The advantage is in delivering a precise method of defining design patterns. In addition, we present a 'patterns wizard', a tool that automates the application of patterns by metaprogramming techniques, <u>*without*</u> *hiding the source code text from the programmer. We conclude with three case studies and two sample outputs that demonstrate how the application of design patterns can be automated using metaprogramming techniques.*

Keywords: Design patterns, metaprogramming, the programmer's apprentice.

1. Introduction

The phrase *pattern language* was conceived by Alexander in his influential composition "*A Pattern Language*" [Alexander 77] that also introduced the concept of *design pattern*. The OO community leaders who tried to import the patterns modus operandi to the domain of software design, have also adopted this phrase. Concerning the research and analysis of design patterns, a pattern language means a "structured collection of patterns," which otherwise might have been designated as a catalogue. To the computer scientist accustomed to the term *language* being reserved to a programming this usage is a source for a cognitive dissonance.

Perhaps to resolve this dissonance, Coplien was inclined to explicitly state that a *pattern language* is not "a programming language in any ordinary sense of the term" [Coplien 94]. Since it is believed that a design pattern cannot be specified in its general form by a formal language, patterns are described by a natural language. A *pattern language* is then a semi-structured prose that comprises a collection of related design patterns, that the software engineer uses as reference to folklore during the process of design.

Contrary to the common belief, we show in this article that a significant part of the body of many design patterns can be specified in a formal language. That is, we show that a substantial part of the procedure prescribed by many design patterns can be expressed in terms of simple manipulations over programs, such as adding an inheritance relation between two classes, adding an instruction to a routine, etc. We suggest a programming language framework used for *precise* specification of design patterns by means of such operations, and also for the manipulation and application of patterns. After showing what precise specifications of design patterns are, we see how a *formal pattern language* can be what is indeed more like "a programming language in the ordinary sense of the term."

Obviously, pattern descriptions reported using a natural language prose are informal, inaccurate, and incomplete, hence the pattern application process is manual, time consuming and error prone. However, the use of a more formal technique has

---

[1] The Department of Computer Science, School of Mathematics, Tel Aviv University, Tel Aviv, Israel. Supported in part by a grant from the German-Israeli Foundation for Scientific Research and Development (GIF).

[2] The Faculty of Computer Science, Technion -- Israel Institute of Technology, Haifa, Israel.

great advantages that need no elucidation, but so is the challenge and the difficulty. This paper is not a complete work but a mere case study of a formal approach to the specification of design patterns. It highlights some related problems, proposes initial solutions, and hopefully inspires further research on the topic.

The term “design pattern” has been used in the literature with various, sometimes conflicting meanings. We restrict ourselves to the interpretation as in [Gamma et. al 95]. According to this interpretation, a design pattern is a description of an explicitly dictated implementation of a solution reusable across similar design problems. The specification of design patterns forms the essence of the design of the suggested solution; the pattern’s description generalizes the implementation detail that did participate in the suggested solution, while all other implementation details are removed.

1.1 PSL: The Pattern’s Specification Language

How should pattern languages be formalized? One possible direction is by using a formal specification language such as Z or Larch to capture the inter-object relationships and dependencies. However, this alone would not have been sufficient: formal specification languages are powerful in describing the external characteristics of any particular system without specifying any implementation details. In contrast, a design pattern is in many cases an implementation strategy that captures a specific solution, and the process of formulating it records the sequence of steps that are taken in its implementation. Formal specification languages were not molded to express implementation details. An even more serious limitation of specification languages is that they are usually not powerful enough to describe a generic family of systems, as it is required in the case design patterns.

Since specification languages are deemed inadequate the approach that we take is to use an ordinary programming language as a leverage for manipulating design patterns. However, instead of manipulating the design patterns directly, the entities manipulated are the programs that are modified to realize these patterns. This is nothing but *metaprogramming*, i.e., writing programs that manipulate programs. The tern *metaprogramming* signifies that there are two separate programming languages combined in a single environment, where one language manipulates the other. The advantage of this approach is that the semantics and the syntax of programming languages are well understood and precisely defined. Further on this section we discuss the disadvantages of this approach, such as the abstraction level of OOPL grammatical constructs being too low.

Using an ordinary OOPL by which the *pattern specification language* (PSL) is phrased subjects it to the syntax and semantics delivered by this OOPL. Doing so also allows pattern definitions be facilitated by the availability of numerous compilers, debuggers, and other development tools. The OOPL that is used for pattern specification is designated the *metalanguage*, i.e., the language in which the manipulation of programs is phrased.

By this approach, a design pattern is formalized by phrasing it as an algorithm in the OOPL chosen as the metaprogramming language. A pattern algorithm rigorously indicates the sequence of steps taken to apply a pattern, hence it is realized as a routine in the PSL. We denote *prescriptive pattern* a design pattern that can be (not necessarily completely, but perhaps for the most part) precisely prescribed in PSL. The Observer pattern (originally appeared in [Gamma et. al 95]) is an example for a prescriptive pattern, and Table 2 lists the steps in the algorithm that realizes it.

The language over which PSL routines operate is denoted the *object language*. The entities manipulated by pattern routines are the object language constructs, such as classes, objects, inheritance relations, aggregation relations, statements, expressions, etc. A design pattern is applied by passing such constructs as arguments to a pattern routine, which modifies them as required.

However, the specification of a design pattern by directly operating over object language constructs is tedious, since even a simple operation such as adding a feature to a class warrants a large number of simpler operations. We can investigate the simple case of inserting a new feature to a class, as an example to how inappropriate this act is being carried out by direct manipulation of low level linguistic constructs. Along with generating the definition of the feature and implanting the actual clause in the class declaration, it is necessary to check the consistency of the it with existing constructs. For example, the new feature’s name may clash with existing features of the same class, parent classes, or child classes. In such cases either overloading or overriding is considered, and the class declaration need be modified as necessary according to the syntax of the object language.

To best handle this problem we identify several abstractions of the object language, each suitable to specify operations of a different level of detail. The description of the implementation in section 2 refers to the different levels used in accomplishing the patterns wizard. This description shows that so far three levels of abstraction are involved in the specification and application of design patterns.

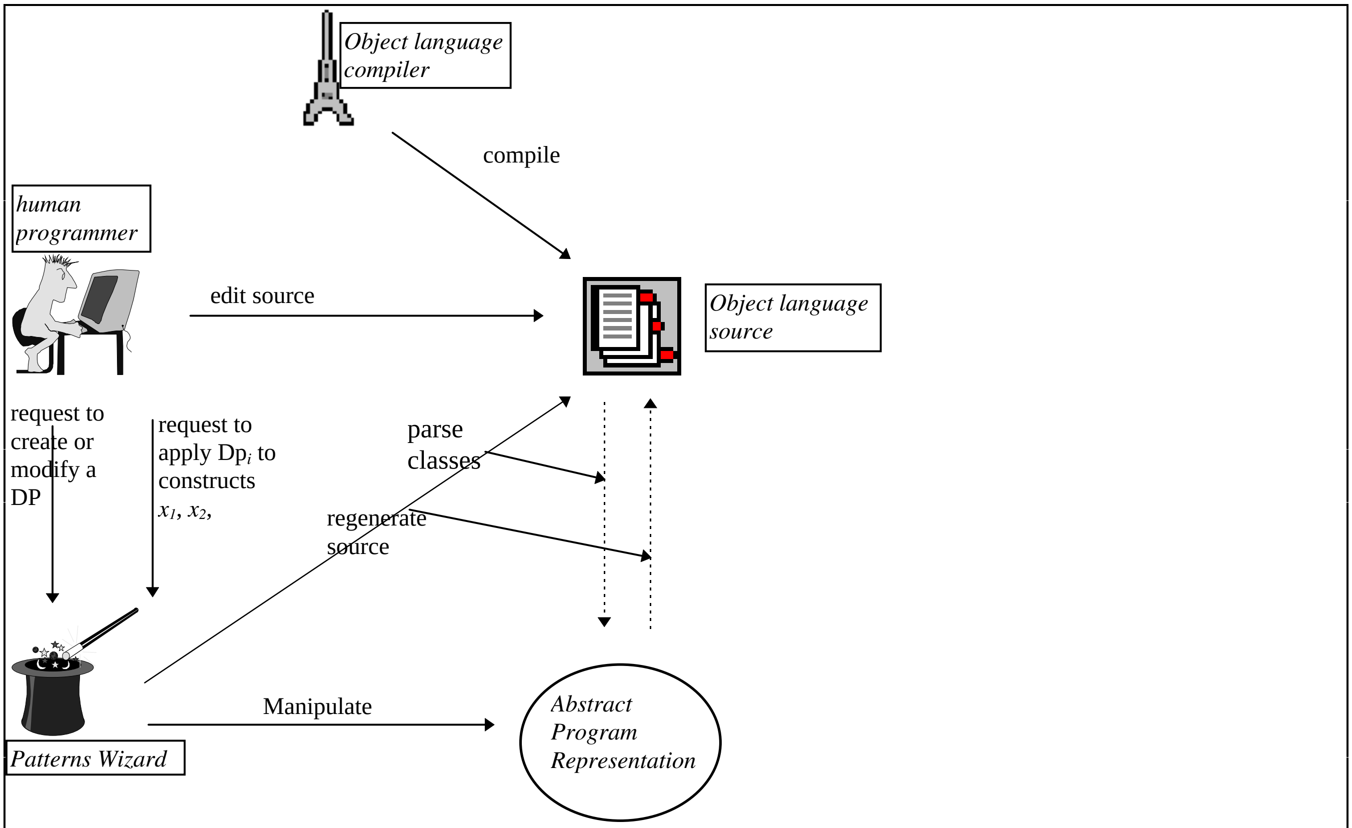

Figure 1: Working with the patterns wizard

1.2 The patterns’ wizard

We implemented a prototype of a tool called ‘patterns wizard' for the specification and application of design patterns. The patterns wizard reads object language code, generates an abstract representation of it in PSL, and can apply pattern routines to concrete constructs of the object language. The wizard is used to semi-automate the application of prescriptive design patterns to existing code, a process which is currently purely manual. Table 1 lists the distinct functions delivered by the pattern wizard, and Figure 1 depicts our vision of software development in collaboration with it.

We should clarify the meaning of the process of pattern *application* performed by the wizard and its objectives. Since PSL is written in an ordinary programming language, the metalanguage, a pattern is implemented as a routine in this language that accepts as arguments elements of programs of the object language and manipulates them to introduce the pattern into the program. Naturally, the manipulated constructs need be available in an abstract form that is convenient to the application of operations by which design patterns are prescribed. The abstract representation need also be compatible with the PSL routines, i.e., formulated in the metalanguage. The form to which the object language is abstracted is referred to simply as “the abstract representation of the object language”; Figure 2 illustrates the abstract representation we chose to implement.

The patterns wizard does (or supports doing) the following:

1. Parse object language text and produce an abstract representation of the program
2. Read, store, retrieve, and modify PSL routines
3. Accept and execute instructions from a human operator that wishes to apply a pattern to suitable constructs
4. Reproduce object language text from the abstract representation

Table 1: Distinct functions delivered by the patterns wizard

Taking the *Observer* pattern as an example (detailed in Figure 3), the related PSL routine takes two class declarations as arguments and makes of the first an "observer" of the second. The sequence of steps that are taken by such routine is listed in Table 2.

It is clear from the last example that even with prescriptive patterns only part of the pattern specification can be formulated by means of simple manipulations of program elements, and therefore can be automated. The application of the rest of the pattern is completed by the programmer manually editing the result after translating it to the textual representation. Thus, the human programmer is assisted by the wizard *in combination with* the contemporary tools that commonly participate in the software development process. As also implied by the separate collaborators illustrated in Figure 1, we do not expect the wizard to be complete with reference to the patterns' semantics, even with reference to prescriptive patterns. For this reason the support gained from the wizard is termed *semi-automation* of the pattern application.

It is also clear that the expressive power of existing OOPLs and other programming languages has its own merits, and higher-level tools cannot replace it; they can only add to it[3]. Taking our example of the *Observer* routine, it is the responsibility of the human programmer to insert calls to the `notify()` routine which triggers a notification about change to the observers. It is also her responsibility to implement the body of the `update()` routine of the numerous *Observer* classes. The automatization of the last two activities is sophisticated beyond the scope of simple metaprogramming, and the complexity of reasoning required to perform them properly is of a different level than the one of the tool discussed here.

For the same reason, we do not seek to automate the *detection*, *recognition*, or *definition* of design patterns at this stage of our research, but only their *application*. We are also aware that many design patterns are not prescriptive in the sense defined above, such as the *Template method* [Gamma et. al 95]. In the case of a number of design patterns such as the *Template* pattern, a major part of the application of this pattern cannot be translated in a straight-forward manner to a metaprogramming algorithm, so that the benefit in using the patterns wizard is minimal. In other cases, such as the *Strategy* pattern, the relations between the agents that collaborate in the implementation suggested by the pattern and the manner by which they cooperate can be described only by general terms that are too abstract to be translated into a metaprogramming algorithm. The *template* pattern is an example for a non prescriptive pattern for the first reason, and the *iterator* pattern is non descriptive for the second reason quoted here. Techniques for handling reasoning of a nature that is "too abstract" for the patterns wizard appear to belong to the realm of artificial intelligence. Their research may benefit from machine learning or knowledge-based techniques.

Outline: The rest of this paper is organized as follows: Section 2 describes the implementation of PSL and the abstract representation of the object language, and also the implementation of the patterns wizard. Section 3 gives an account of two case studies in translating prescriptive patterns to PSL routines, and an example how it can be used to enforce design principles. Finally, section 3.3 gives the conclusion and delineates the future research.

2. Implementing PSL and the Wizard

There is a finite number of recurring formations, such as relations and procedures, that form the building blocks of design patterns. These elements may be viewed as rudimentary design patterns themselves. The definition of a PSL is in fact a definition of a compact collection of building blocks that jointly are sufficient to specify the steps that may be automated in the application of any prescriptive design pattern.

It may seem that a complete definition of such collection requires a universal catalog of design patterns. In the absence of such improbable catalog, the recording of patterns building blocks is instrumented by enumerating (an abstracted form of) the constructs of the object language and the necessary operations that apply to each. It follows that a careful observation of prescriptive patterns at hand, in combination with a suitable level of abstraction over the object language, can lead to PSL definition sufficient for the specification of numerous design patterns. Therefore the expressiveness of the implementation suggested here is ultimately tested empirically, that is, by using it to specify prescriptive patterns that were not considered in the case studies reported in section 3.

---

[3] Learning from experience with past abstraction mechanisms, such as 4GLs (4th-generation languages), initial CASE tools, code generators and application generators, it appears that when high-level constructs assumed the place of the old-fashioned programming language, the result is always a language of a expressive power limited to a very specialized domain.

2.1 Selecting the implementation languages

At least two OOPLs are considered to use in the definition of PSL and in the wizard's implementation: The metalanguage, by which PSL is defined, and the object language, the text of which is analyzed. It follows from the above discussion that the selection of each language has far reaching implications on the expressiveness and ease of use of the final product. We hereby concisely present the arguments that led us to the choices we made.

For each implementation we first considered the issue of whether a statically typed ("static") or a dynamically typed ("dynamic") language should be used. As for the object language, static languages have the advantage of statically having type information available, which dynamic languages miss[4]. Also, a language is suitable to serve as the object language if an application of the wizard significantly reduces the development effort in terms of textual changes, consistency validation, and managing the technical aspects of the language. Our pick was Eiffel as the first choice for a host of benefits it has. The language has a simple and well-defined grammar, and its syntactical constructs are close enough to the elements of abstract information used by the PSL to allow convenient transformation. Also, a suitable Eiffel parser was available that is described in detail in section 2.4.

Given the different functions of the wizard as listed in Table 1, the metalanguage by which PSL is defined is preferably a dynamic OOPL. Smalltalk, supported by an environment such as the Smalltalk/V and a library of useful container classes and formatting tools, is convenient for rapid prototyping of the PSL and generating the abstract representation of Eiffel. The advantages of dynamic languages in general and Smalltalk in particular are being powerful and flexible, while performance and safety are not a major issue in this stage.

The discussion in implementation languages did not refer to a language or framework of pattern application. The Smalltalk/V environment is also suitable for the application of the PSL, i.e., the library of pattern routines of different levels of abstraction.

The distinction between the object language and the metalanguage implies that the framework we suggest is not necessarily reflective. If, for example, Smalltalk was chosen as the object language the framework might have been considered reflexive. However, we discriminated between the metalanguage from the object language also to stress the fact that by *metaprogramming* characterization of the system it is applicable to any OOPL.

2.2 A sample wizard session

To summarize the implications of the design decisions we took on the structure and performance of the wizard and its related framework we hereby list a sequence of operations that is a characteristic session of patterns wizard exercise:

1. Eiffel source is parsed and an abstract representation is generated. The syntactic analysis is performed using an Eiffel parser based on the TROOPER [Avotins, Maughan & Mingins 95] program, as described in section 2.4. TROOPER produces a set of trees: each tree represents the syntactical structure of a single Eiffel class, by holding a single (Smalltalk) object for every syntactical construct in the Eiffel grammar [Meyer 94].

2. The syntactic representation is abstracted to the level described in section 2.3, which is more suitable for the description of the elements of design patterns. Clearly this form is more convenient for the specification of the pattern routines.

3. The programmer applies a pattern by executing the associated routine to the selected classes and constructs as arguments. If the actual arguments of the executed routine conform to the interface defined and if there were no inconsistencies between the result and the rest of the object program, then the abstract representation is updated accordingly.

   If the user is not satisfied with the pattern specification, he may correct it by modifying its routine or add to its functionality. Thanks to the dynamic nature of the Smalltalk environment this objective is ever so easy to accomplish without disrupting the work flow.

4. In order to let the programmer complete manually the fraction of the pattern that was not automated the system produces a textual representation of the classes that were directly affected by the last operation. Special notes may

---

[4] Although at some stage Smalltalk was also considered and was found a suitable candidate, in particular because of the availability of abstract metalinguistic information from Smalltalk systems. Smalltalk is not suitable for the reasons that follow.

be made as references to the locations where modifications were introduced; other notes may refer to the places where definitions must be completed, and so on.

5. The programmer iterates through step 3 in order to further enhance and fine-tune the library.
6. Eiffel source of the complete library is generated to replace its previous version.

2.3 The abstract representation of the object language

The exact form by which the object language is abstracted bears significant implications on the PSL since it determines the pattern language's 'nouns and verbs'. For example, because the inheritance information included in the class declaration is represented using a descendant of the `OrederedCollection` class, a new inheritance relation is attached to a class using any version of the method `add:` that adds a new member to a collection. Hence it is possible to use all available variations on `do:` , such as `collect:`, `select:`, etc., and the entire interface of class `OrederedCollection` in regard to manipulation of the inheritance relations.