Precise Specification and Automatic Application of Design Patterns[i,ii,iii]

Amnon H. Eden, **Amiram Yehudai**
Department of Computer Science,
School of Mathematics, Tel Aviv University,
Tel Aviv, Israel
{eden,amiram}@math.tau.ac.il

Joseph (Yossi) Gil
Faculty of Computer Science,
Technion -- Israel Institute of Technology,
Haifa, Israel
yogi@cs.technion.ac.il

Abstract

Despite vast interest in design patterns, the specification and application of patterns is generally assumed to rely on manual implementation. We describe a precise method of specifying how a design pattern is applied: by phrasing it as an algorithm in a meta-programming language. We present a prototype of a tool that supports the specification of design patterns and their realization in a given program. Our prototype allows automatic application of design patterns without obstructing the source code text from the programmer, whom may edit it at will. We demonstrate pattern specification in meta-programming techniques and a sample outcome of its application.

Keywords: Application of design patterns, tool support for design patterns, metaprogramming

1. Introduction

Today, design patterns form an object oriented design heuristic of prime importance. Patterns form an excellent method of conveying experience from the expert to the novice, and allow the concise communication of the essentials of the design process and its results. Readers not familiar with design patterns should consult [13, 14], and in particular [12].

Nonetheless, design patterns are mostly communicated using natural language narrative. The use of informal language is often deemed inevitable: “a solution, the essence of a pattern, … transcends the exact nature of its expression [1]”; thus, one is lead to suppose “a pattern must ultimately reside in one’s own mind. [Ibid.]” Even so, does “ultimately” implies “exclusively”? Informal specification is deficient as it leads to repeated manual programming, which is error prone, unsafe, and time consuming.

We looked for a method of precise formulation of the solution a design pattern specifies. Contrary to our purpose, which is to automate the implementation of design patterns, several previous formalization methods were intended to enforce design decisions dictated by patterns. For instance, Helm, Holland and Gangopadhyay [2] propose a language of contracts that specifies the implementation of collaborators in a “behavioral composition” and imposes restrictions on their identities and the valid sequence of operations. Similarly, Partha [3] demonstrates how the behavior indicated by a pattern can be translated using the LGA concept into regularities to enforce the obligations each collaborator takes. These regularities (or “rules”) are predicates that express global rules governing relations between modules. The rules can be imposed using the *Darwin-E* environment [4] which detect violations of such predicates by means of static analysis. Another similar work by Klarlund & Koistinen [5] presents a first order logic language of parse trees, *CDL*, that can be used to validate design constraints. *CDL* predicates are proposed as means to express constraints over the behavior and relations of collaborators in a design pattern.

Bosch [6] offers a constructive approach at the realization of design patterns. He proposes a new OOPL: *LayOM*, with an extended object model that translates to C++. The language supports uncommon object properties such as *layers*, *categories*, and *states*, which

[i] Also available at: `http://www.math.tau.ac.il/~eden/precise_specification_and_automatic_application_of_design_patterns.{ps.Z,rtf.zip}`

[ii] Supported in part by the German-Israeli Foundation for Scientific Research and Development (GIF) and by the Israeli Ministry of Science.

[iii] Published in “Automated Software Engineering - ASE'97”

prove useful means to express the roles of different collaborators in a design pattern. *LayOM* offers an elegant specification of design patterns that preserves their identity, but it is highly specialized and does not form a general solution. Alencar, Cowan and Lucena [7] use temporal logic to describe relations among components in terms of category theory. The authors, however, do not explain how exactly their language expresses the diverse possible relations between constructs of OOPLs, nor discuss mechanizing the construction of the patterns.

How can the application of a design pattern be supported best? In their article, Budinsky, Finnie, Vlissides, and Yu [8] present a tool that supports the application of patterns by generating code according to the specification of the programmer, while ignoring existing code. This approach assumes that patterns involve classes dedicated to their role as collaborator within a particular design pattern. This is not true in general: patterns rarely exist in isolation. More often it happens that a *collaborator* in one pattern plays a role in another[i]. The definition of a pattern stresses this fact: a pattern is "*a solution to a problem in* ***a particular context***" [9, 10]. Moreover, a pattern is implemented not by merely producing new classes and routines, but more often by adapting the existing context, i.e., preexisting program constructs, to the roles they assume in the newly applied pattern. We conclude that a more beneficial (and realistic) application of a design pattern should adapt existing constructs in a program, a task that is more complex and context-dependent than brute code generation.

Another possible direction is by using a formal specification language such as Z or Larch to capture the inter-object relationships and dependencies. Nevertheless, 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. By contrast, a design pattern is most often an implementation strategy that captures a specific solution, and the process of formulating it directly refers to the steps that are taken in its implementation. Formal specification languages, however, were not molded to express implementation details. An even more serious limitation of specification languages is that they are not powerful enough to describe a generic family of systems, as required in the case of design patterns.

By contrast, our metaprogramming approach is rooted in the fact that the verbal specification of a well defined design pattern readily transforms into a *trick* (defined in section 2): an algorithm that manipulates elements of programs, whose execution realizes the respective lattice, as demonstrated in sections 3 and 6. In comparison, clichés [11] allow one to encapsulate a generalization of particular computing operations, such as *table-lookup*, or *linear search*; a cliché serves as a blueprint to the program synthesizer, which specializes it as necessary. *Tricks* (defined below), however, manipulate programs directly, thus operate on a level higher than the programming language, that of the metalanguage.

The rest of this paper is organized as follows: Section 2 discusses the benefits of the metaprogramming approach and the roles of a tool supporting the implementation of lattices. Section 3 discusses the limitations of the metaprogramming approach. Section 4 discusses micro-patterns and the prospects of recognition and discovery of design patterns. Section 5 elaborates on the prototype of the patterns wizard. Section 6 demonstrates the metaprogramming approach using sample tricks and the enforcement of a design principle using the wizard. Section 7 concludes with a discussion of future directions.

2. The Metaprogramming Approach

In this section we discuss the elements and terms in the metaprogramming approach and elaborate on the roles of a tool that support in the implementation of design patterns.

Different pattern writers use any one of a number of prevalent forms [12, 13, 14], all of which include a statement of a *problem* (*applicability* in [12]) and an abstraction of an appropriate *solution* (*structure, participants,* and *collaborations* in [12]) that is plausible in the context specified. By contrast, the mechanization technique presented here focuses in the *solution* element of design patterns [15][ii].

Definition: To distinguish it from the other meanings of 'pattern', we designate the general layout of structure and behavior specified by the solution dictated by a design pattern as the *lattice* of this pattern.

We focus on tool support for the implementation of lattices[iii]. Such tool forms an environment that supports the definition and execution of operators (routines) in a

[i] In particular because software development, in contrast with the waterfall model, often takes the form of repetitive refinement, expansion, consolidation, and gradual evolution [Booch 94; Gamma et. al 95, p. 354].

[ii] Please refer to [15] for a detailed justification.

[iii] Please refer to section 4 for the prospects of recognition and discovery of design patterns.

pattern specification language[i] (PSL), and the facilities required for their execution in a particular context. In section 5 we present a prototype of such tool, the *patterns wizard*.

2.1 Intended Contributions

Unlike previous attempts at the formalization of pattern specification, the metaprogramming approach combines the following qualities:

1. It achieves **safety** and **consistency**, as the implementation is specified in precise terms
2. It **speeds the implementation**, as it automates much of the coding and the reasoning involved
3. **New code is integrated** into the existing program in the process of implementing a lattice. Unlike [8], this approach operates in the pre-existing context of a given program, and may modify elements thereof as needed.
4. Because existing code is processed, it is useful for introducing design patterns to **legacy systems** and not only in developing new programs (see also the note on the recognition of patterns in section 4)
5. A tool supporting this approach need **not interfere with source code editing** by the programmer, which may take place at any point in the process
6. It only allows **precise generalizations**, which preserve the generality of the pattern but eliminate ambiguities possible with contemporary techniques of pattern specification, which blur the difference between one design pattern and another.
7. Unlike specialized formal languages, the *pattern specification language* is **expressive** enough to allow direct manipulation of any semantically significant construct in the implementation process.
8. It is useful also for **validation of design decisions** (as demonstrated in section 6.3)

We have chosen to use an ordinary programming language as a leverage for manipulating design patterns. In other words, the algorithms that introduce a lattice into a given program (designated below as *tricks*) are realized in an ordinary OOPL. The advantage of using an OOPL as a PSL is that the semantics and the syntax of programming languages are well understood and (generally) defined. Another advantage in using common PLs is them having mature development tools (such as debuggers and browsers) that are useful throughout the development process. This is nothing but *metaprogramming*, i.e., writing programs that manipulate other programs.

[i] Thus we demonstrate how a *formal pattern language* can be a programming language "in the ordinary sense of the term," as opposed to the Alexandrine originated usage of "pattern language", which is not "a programming language in any ordinary sense of the term [Coplien 94]."

2.2 Terminology

The language that is used for the specification of lattices, i.e., the language in which the manipulation of programs is phrased, is the *metalanguage*. The *pattern specification language* (PSL) results from complementing the metalanguage with the *abstract syntax* (see below), which models the *object programs*.

The language over which PSL routines operate is called the *object language*. An program in the object language processed in a particular session is the *object program*.

The term *metaprogramming* signifies that the metalanguage and the object language are two (possibly different) programming languages combined in a single environment, where one is used to manipulate the other. Note that the metaprogramming framework is not necessarily reflexive.

The syntactic elements of the object language were too detailed for our purpose, as the specification of lattices by manipulations syntactical constructs directly, even in the simplest cases, involves a large number of syntactic constructs. Thus we introduced the *abstract syntax*, a modeling mechanism supplementing the metalanguage, for representing the semantically significant elements of programs in the object language (such as classes, objects, inheritance relations, aggregation relations, various statements and expressions, etc.) We defined the abstract syntax also in order to specify lattices in a *language independent* manner[ii]. Our intention was that whenever the same lattice applies to two different OOPLs, the respective PSL routine (*trick*) will be valid to both. The abstract syntax serves this purpose by abstracting the grammatical detail into a syntax-independent form.

The *internal representation* is an instance of the abstract syntax generated to model a particular object program.

[ii] That is, at least as much as it is possible that a particular design pattern is language independent. Obviously a pattern that is useful in one PL may be trivial within the second (being available as a built-in feature of this language), and at the same time impossible in a third PL (employing features that are missing in this language).

A *trick* is an operator specifying the sequence of steps taken in the realization of a lattice. It does so by adapting designated elements of the abstract syntax and possibly by introducing new elements or by deleting existing ones. The *implementation of a trick* is a routine in the metalanguage with formal arguments of the *abstract syntax*, which carries out that operator through manipulations defined over them.

To *apply* a trick (or the respective lattice) means to execute the PSL routine that implements this trick with designated elements of the object program passed as the actual arguments.

2.3 Tool Support for the Specification and Application of *tricks*

Before we describe our *patterns wizard*, a prototype for a tool that supports the specification and application of *tricks* by the principles of the metaprogramming approach, we wish to explain first what is generally expected from such tool. In section 5 we outline how the requirements mentioned below were met by the wizard's.

To begin with, an automating tool maintains a library of routines, each of which implements a particular *trick*, and supports their modification, application, and debugging.

Furthermore, because PSL routines operate over abstract syntax elements (instead of the object program's text or a syntactic representation thereof), the object program must be transformed into a representation in the abstract syntax. The object program need not necessarily be transformed in one step, but we require that different classes can be parsed in different stages of the development cycle and integrated incrementally into the internal representation of the object program. Table 1 summarized the responsibilities of a pattern automation tool.

Most importantly, we recognize the need of the programmer for manual editing of the object program's source code. Clearly the expressive power of existing OOPLs has its own merits, and higher-level tools cannot replace it, but only add to it[i]. For this reason it is required that the programmer will also have direct access to the source code. To emphasize this point, Figure 1 depicts a vision of software development in collaboration with the patterns wizard, our prototype for a support tool.

[i] Learning from experience with past abstraction mechanisms, such as 4GLs (4th-generation languages), early CASE tools and code generators or application generators, it appears that if higher-level style assumed the place of the old-fashioned PL the result is always a language of a expressive power limited to a very specialized domain.

3. Analysis of Informal Specification

We should clarify what aspects of the implementation of lattices are *not* addressed by any formal approach in general, and the metaprogramming technique in particular.

Given the commonly used informal description of lattices, only the precisely described elements of a lattice can be successfully translated into a precise specification, and possibly automated. Fragments of the common, informal descriptions of each lattice fall into exactly one of the following categories:

1. Precise, singular specification (such as "a routine named 'update' with no arguments")
2. A list of alternative interpretations (such as "A class adapter uses multiple inheritance to adapt one interface to another: … An object adapter relies on object composition [12, p. 141]", or "*x* could take *y* as an argument")
3. A specification that can be subjected to precise generalization (such as "a routine with $n > 3$ arguments", etc.)
4. A specification using a technical term, where there is more than one valid interpretations (such as "*x* caches information about *y*")
5. Intentional omission of detail, where (possibly infinitely) many interpretations are valid; for instance, the specification of the *Observer* pattern [12] ignores how the information about the change is conveyed from the subject to the observer. For instance, see in [16] different specialization of the pattern.
6. [Intentionally?] Fuzzy or abstract description that cannot be readily translated into simple metaprogramming commands. The reason may vary, such a specification by a role ("ConcreteStrategy <u>implements the algorithm</u> using the strategy interface. [12, p. 317]")

As demonstrated in section 6, our analysis revealed that while the few patterns whose essentials fall into the last category are deeded impossible to automate, many design patterns benefit from our approach.

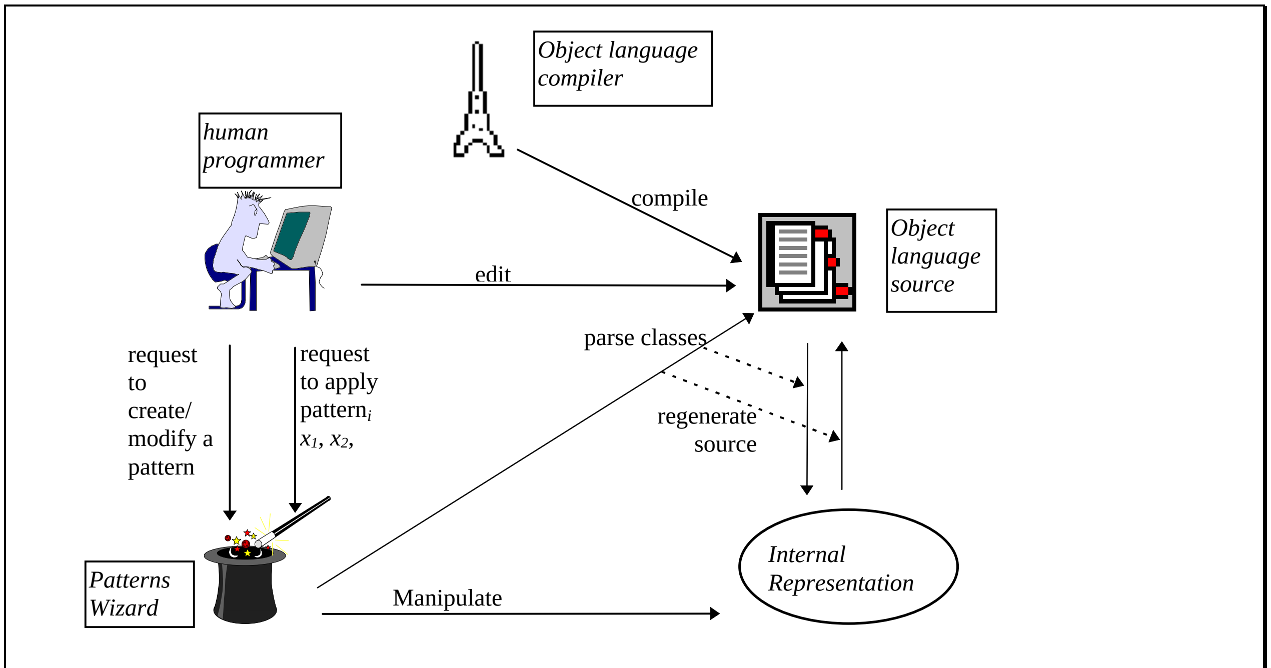

Figure 1: Working with the patterns wizard

1. **Abstract**: Parse object language text and produce an internal representation
2. **Maintain** the **abstract syntax** representation, possibly as a class library in the metalanguage
3. **Maintain a tricks 'library'**: support operations such as read, store, retrieve, and modify PSL routines
4. **Apply**: By the user's command, execute a pattern routine over designated arguments
5. **Transcribe**: Generate a textual representation of the current object program from the internal representation as source code in the object language

Table 1: The responsibilities of a pattern automation tool

Original text	***Precise form***
"Proxy: maintains a reference that lets the proxy access the real subject …; provides an interface identical to subject's …; forwards requests to RealSubject"	The Proxy trick accepts a `subject` class as argument. It generates a `proxy` class with a reference attribute to `subject`. The `proxy` class overrides each of the abstract routines of the `subject` class with a concrete routine, that forwards the call to the `subject`.
"Virtual proxy may cache additional information about the real subject so that they can postpone accessing to it."	The CacheProxy trick accepts yet another argument: `cached`; the user passes a subset of the `subject`'s attributes as `cached`. For each cached attribute att$\in$`cached`, the generated `proxy` class shall have a duplicated attribute att', and respective inspector (routine) that returns att' instead of accessing `subject`. Add a `refresh` routine which for each att' assigns it with the respective attribute att of the subject: att' := `subject`.att; invoke `refresh` when the `subject` reference is initialized.

Table 2: Segments of the formalization of the Proxy *pattern [12, pp. 209-210]*

Because generalizations of categories 5 and 6 are unavoidable parts of pattern specifications, there is no choice but to leave some of the implementation to the programmer. The wizard can assist in pointing out the changes it induced and the additional changes required.

By contrast, when various interpretations appear valid, we acknowledge three alternative choices for the wizard's operator:

1. Omit the "volatile" part and let the programmer complete it manually
2. Provide one routine that follows an interpretation of interest which the user shall stick to
3. Provide several alternative routines, each of which follows an interpretation of interest

Table 2 gives a detailed example for how an informal specification is treated.

4. Micro-Patterns

Precise specifications of lattices promote the investigation of numerous important aspects of the use of design patterns, such as: How lattices combine and interact? How can design patterns be indexed and classified in search-effective format?

Of equal importance are the questions of *recognition* and *discovery*, namely: Is it possible to trace the occurrence of a predefined lattice in legacy code? Is it possible to discover novel lattices of interest and significance? Despite previous attempts at formal specification of design patterns, possibly also used to recognize lattices in programs, neither of these attempts fit with a constructive scheme nor deals with discovering lattices of interest.

Recent advances in our research lead us to the following observations:

1. A concise set of smaller-scale design motifs, designated as **micro-patterns,** is sufficiently expressive as playing a repeated part in the specification of (the tricks of) numerous design patterns.
2. Micro-patterns (and their respective tricks) form an intermediate level of abstraction, lower to design patterns and above the PSL.
3. The **recognition** of the application of a trick amounts to tracing the result of its execution in a given program and inferring the actual arguments used. Initial analysis of existing tricks reveals that this process is at least decidable, although its complexity has not been established yet.
4. The **discovery** of a lattice of interest can be interpreted as discovering "interesting" combinations of the lattices of micro patterns; as such, *discovery* amounts to the recognition of such "valid" combinations.

5. The Patterns Wizard

This section details some design decisions in our implementation of a prototype of a tool supporting the implementation of lattices by the metaprogramming approach: the *patterns wizard*.

5.1 Programming Languages

Two OOPLs are involved in the specification of lattices: the metalanguage and the object language. For the choice of each language we first considered the issue of whether a statically typed ("static") or a dynamically typed ("dynamic") OOPL should be used. As for the object language, static programming languages have the advantage of having type information available, which dynamic languages miss. With a suitable object language a metaprogramming tool can reduce significantly the development effort, manifested in textual changes, consistency validation, and managing the technical aspects of the programming process. Our selection was Eiffel, which also has a simple and well-defined grammar.

Given the wizard's expected functions listed in Table 1, the metalanguage by which PSL is defined is preferably a dynamic OOPL. Smalltalk-80, supported by the Smalltalk/V environment, is convenient for rapid prototyping of the PSL and for generating the internal representation of the object language. The advantages of Smalltalk are its expressive power, its powerful class library, and its flexibility, while performance and safety are secondary.

Although this choice subjects our PSL to Smalltalk's syntax and semantics, it does not equate our approach with mere 'Smalltalk metaprogramming.' Our metalanguage of choice in implementing the wizard's prototype serves to demonstrate the expressiveness and usefulness of the metaprogramming approach.

Eiffel parsing was implemented by TROOPER [17]. The parser produces a Concrete Syntax Tree (CST), implemented as a compound Eiffel object comprising of the respective linguistic constructs.

5.2 A Sample Session of Pattern Application

Following is a characteristic pattern application session supported by the wizard. The development process is assumed to have reached the implementation stage and the user has just decided to apply certain design patterns along with some initial constructs[i]. We also assume the existence of a patterns' library that contains the implementation of the required tricks.

I. The object program text is parsed and an internal representation is generated. Parsing involves either the entire system or only a subset thereof.

II. A pattern is selected and the respective routine is executed over designated elements of the object program, if any.

   A. If the predefined restrictions over its arguments hold then the routine concludes successfully and the pattern is realized; otherwise, an error message is produced.

   B. If the user is not satisfied with the results of the application of some routine than she does the following:

      ♦ 'undo' the routine's execution, and

      ♦ correct its body as desired

   C. Thanks to the dynamic nature of the Smalltalk environment, changing dynamically the specification of a certain routine is ever so easy to accomplish without disrupting the work flow.

   D. The user repeats these steps until the desired result is achieved, or manual editing is required.

III. The system transcribes the source code of the object program, annotated with notes directing the user to the locations where manual programming is required.

An advanced revision of the wizard can allow source code editing simultaneously with the existence of the internal representation, for instance. We do not discuss such features here as their implementation does not seem to pose any theoretical challenges of particular interest.

6. Case study

Using the metaprogramming technique, we implemented an approximation of the lattices of the following patterns [12]: *Abstract Factory*, *Class* and *Object Adapter*, *Proxy* and *Cached Proxy*, *Decorator*, *Recursive Composite*, *Visitor*, *Observer*, and *State*. This section demonstrates wizard-style metaprogramming through the specification of the *Cache-Proxy* design pattern and the enforcement of a design principle. Other examples appear in [18].

[i] For convenience, the wizard may support brute code generation, such as "a class in a click" to generate a class stub, etc.

6.1 The *Cache-Proxy* Design Pattern

We present the specification of the *Cache-Proxy* version of the *Proxy* pattern [12] in a pseudo-code format, whose Smalltalk implementation appears in [18]. The specification employs several micro patterns, some of which are also detailed below. Verbs that appear in capitalized italics (*FORWARDING*) designate a trick that introduces a [micro] pattern.

I CACHE-PROXY (design pattern)

Create a proxy class which *COUNTERFEITS* some of the features of a `realSubject` and *CACHES* the rest.

Uses: *ABSTRACTION*, *MATERIALIZATION*, *INTEGRATION*, *COUNTERFEIT*, *CACHE*, *REPLYING*

Synopsis:

```
approximate: realSubjectCls
     cache: features2Cache
```

(Returns: new proxy class)

Semantics:

```
1. create proxyCls by MATERIALIZING subjectCls
2. let r designate the result of INTEGRATING the realSubjectCls in the proxyCls (the result of which is new attribute)
3. create a refresh concrete routine in proxyCls with no arguments, return value is realSubjectCls.
4. in refresh check if the ASSOCIATION to the realSubjectCls is set; if not, INITIALIZE it.
5. for each feature f∈ realSubjectCls do:
       if f∈ features2Cache
           then CACHE f in proxyCls using r, refresh routine is refresh
       else Counterfeit f in proxyCls using refresh.
6. apply REPLYING to refresh with r
7. for each creator of proxyCls add a call to refresh
```

II ASSOCIATION (micro-pattern)

Create an association (reference) from `fromClassdec` to `toType`.

Synopsis:

```
associate: fromClassdec with: toType
```

(Returns: the new attribute).

Complete specification omitted.

III CACHE (micro-pattern)

Cache in `cacheCls` the value of `aFeature` in `sourceObject` by (1) creating an cached attribute initialized within `aConcreteRoutine`, and (2) creating (or adjusting) an inspector that returns the cached value.

Synopsis:
```
cache: aFeature
     in: cacheCls
     through: throughObject
     refreshRoutine: aConcreteRoutine
```
(Returns: the function in `cacheCls`)

Complete specification omitted.

IV COUNTERFEIT (micro-pattern)

To a given class, add a concrete routine `aFeature` that *FORWARDS* the call to an *INTEGRATED* object (or modify an existing routine by this name to do so).

Used by: *DISGUISEMENT*, *CACHE-PROXY*

Uses: *FORWARDING*

Synopsis:
```
     counterfeit: aFeature
             in: cfClassDec
             using: anObject
```
(Result: the old/new routine)

Semantics:

1. **if** a feature called "`aFeature`" exists in `cfClassDec` **then**

   if `aFeature` is an abstract routine
   then change it to a concrete routine *r* which *PORTRAYS* its previous version
   else if `aFeature` is a concrete routine
   then let *r* designate this routine
   else (`aFeature` is an attribute in `cfClassDec`) produce an error message

   else create a concrete routine *r* which *PORTRAYS* `aFeature` as a concrete routine (`portrayAsConcrete`) and add it to `cfClassDec`
2. *FORWARD r* to *a*.

V FORWARDING (micro-pattern)

Forward a routine call through `anExpObject` to a routine by the same name and arguments.

Used by: *COUNTERFEIT*

Uses: *REPLYING*

Synopsis:
```
forward: aRoutine through: anExpObject
```
Semantics:

1. create a call *c* to `aRoutine` qualified by `anExpObject` with the actual arguments of `aRoutine`
2. add *c* to the body of `aRoutine`
3. apply *REPLYING* of *c* in `aRoutine`

VI INITIALIZATION (micro-pattern)

Initialize `ref` using `unqualifiedCall` as a constructor.

Synopsis:
```
initialize: ref
     calling: unqualifiedCall
```
(Returns: the new instruction)

Complete specification omitted.

VII INTEGRATION (micro-pattern)

ASSOCIATE `wholeClass` to an object of `partCls`. *INITIALIZE* the reference to `partCls` in creation of `wholeClassDec` with a new, dedicated object.

Synopsis:
```
Integrate: partCls in: wholeClass
```
Complete specification omitted.

VIII MATERIALIZATION (micro-pattern)

Create a class called `childClassName` that derives from `parentCls`, and define a concrete routine in the child for every abstract routine of `parentCls`.

Synopsis:
```
materialize: parentCls
     designated: childClassName
```
(Returns: the new child class)

Complete specification omitted.

IX REPLYING (micro-pattern)

Add a new statement to `aConcreteRoutine` that gives the result of computing an expression as a function return ('result') statement.

Synopsis:
```
reply: anExpression in: aConcreteRoutine
```

6.2 Sample Output

The following sample code of the result of executing the *CACHE-PROXY* trick. It consists of one manually created Eiffel class, `IMAGE`, and two other classes generated by the execution of `approximate:cache:` with `IMAGE` passed as "real subject" and the first two features of `IMAGE` (`file_name` and `extent`) serving as the set of features to be cached. The `IMAGE_SUBJECT` class was generated by yet another trick: *Abstraction*, applied to `realSubjectCls` in advance, to serve as the subject (which defines the common interface for `IMAGE` and `REAL_SUBJECT`). `IMAGE_PROXY` is the proxy class. Modifications implanted by the wizard are typeset in bold letters.

```
class IMAGE inherit
   IMAGE_SUBJECT
      redefine
         file_name,extent,data,pixel_at
      end
feature
   file_name: STRING;
   extent: POINT;
   data: TWO_DIMENSIONAL_ARRAY[PIXEL];
   pixel_at(x:INTEGER,y:INTEGER):PIXEL is
   do
      Result := data.at(x,y)
   end
end -- class IMAGE
```

```
deferred class IMAGE_SUBJECT
feature
   file_name: STRING is
      deferred
      end;
   extent: POINT is
      deferred
      end;
   data: TWO_DIMENSIONAL_ARRAY[PIXEL] is
      deferred
      end;
   pixel_at(x:INTEGER,y:INTEGER):PIXEL is
   deferred
   end
end -- class IMAGE_SUBJECT
```

```
class IMAGE_PROXY inherit
   IMAGE_SUBJECT
      redefine
         file_name,extent,data,pixel_at
      end
creation
   make
feature
   make is
      do
         !IMAGE!associated_image;
         refresh
      end;
   -- Cached attributes
   file_name: STRING is
      do
         Result := cached_file_name
      end;
   extent: POINT is
      do
         Result := refresh.extent
      end;
   -- Non-cached attributes
   data: TWO_DIMENSIONAL_ARRAY[PIXEL] is
      do
         Result := refresh.data
      end;
   pixel_at(x:INTEGER,y:INTEGER):PIXEL is
   do
      Result := refresh.pixel_at(x,y)
   end;
feature {NONE}
   associated_image: IMAGE;
   refresh: IMAGE is
      do
         if associated_image = Void then
            !IMAGE!associated_image
         end;
         cached_file_name :=
              associated_image.file_name;
         Result := associated_image
      end;
    -- cached values:
    cached_file_name: STRING;
    cached_extent: POINT
 end -- class IMAGE_PROXY
```

6.3 Enforcing Design Principles

The patterns wizard can be used to verify adherence to design principles within class libraries of large scale. We found that PSL is also suitable to state design principles formally and explicitly with little effort, and thus test their realization by the wizard. Following is an example of how to verify adherence to the Law of Demeter.

The *Law of Demeter* [19] is a design principle that attempts to reduce the dependencies between classes by restricting calls to access features of “closely related” classes. It is believed that the complexity of the class library is better controlled by enforcing this law.

Concisely, the classes that ”closely relate” to routine *R* of class *C* according to the Law of Demeter, are *C* itself, its parents, the classes that are “subparts” of *C* (its attributes), the declared types of the arguments of the routine *R*, and the classes of the objects locally declared in *R*.

7. Future Directions

There are numerous other ways to employ a metaprogramming tool for the benefit of software development and testing that are not directly related to the research of design patterns. The case study described in section 6.3 shows how design principles can be stated formally in PSL and enforced using the patterns wizard.

One central issue that was not addressed in this article is the question of *traceability*, that is, how is the identity of a lattice, once implemented, preserved as such? This is a desired quality as it supports an abstract view of the system both in terms of granularity and of human-oriented reasoning. Such a scheme may also allow direct command of each lattice instance existing in a given object program past the point of its implementation. Thus an important extension to the current prototype would register each step in the application of the wizard as a “new revision” of the object program, similar to a revision control system, tracking the applied trick and its actual arguments instead of the mere result.

Additional directions for the development of the patterns wizard are the following:

1. Enhance the patterns wizard to other object languages (such as C++ and Smalltalk). This requires investigation of the supposed “language independence of design patterns” quality.
2. Build an interactive tool that encapsulates the patterns wizard in “user friendly," possibly graphic, interface. For example, an interactive

tool can guide its user to the locations in code where the application of the pattern needs to be completed manually.

Acknowledgments

We are grateful to Karl-Heinz Sylla for careful reading of an earlier draft, Dr. Christopher Landauer of for remarks, Kyle Brown for his comments, and Prof. Douglas C. Schmidt for his remarks.

References

[1] Ralph J. and W. Cunningham (1995). “Introduction”, in: John M. Vlissides, James O. Coplien, and Norman L. Kerth (eds.). *Pattern Languages of Program Design 2*, Addison-Wesley.

[2] Helm R., I. M. Holland, and D. Gangopadhyay (1990). “Contracts: Specifying Compositions in Object Oriented Systems". *Proceedings of OOPSLA*, SIGPLAN Notices, vol.25 no.10.

[3] Pal P. P. (1995). “Law-Governed Support for Realizing Design Patterns”. *Proceedings of TOOLS USA* 17.

[4] Minski N. H. (1994). “Law Governed Regularities in Software Systems”. Technical report LCSR-TR-220, Rutgers University, LCSR, Jan. 94.

[5] Klarlund N. and J. Koistinen (1996). “Formal Design Constraints”. *Proceedings of OOPSLA.*

[6] Bosch J. (1996). “Language Support for Design Patterns”. *Proceedings TOOLS Europe '96.*

[7] Alencar P. S. C., D. D. Cowan, C. J. P. Lucena (1996). “A Formal Approach to Architectural Design Patterns”. *Proceedings of the 3rd International Symposium of Formal Methods Europe*, pp. 576-594.

[8] Budinsky F. J., M. A. Finnie, J. M. Vlissides, and P. S. Yu. “Automatic code generation from design patterns”. *Object Technology* Vol. 35, No. 2.

[9] Alexander C. (1979). *The Timeless Way of Building.* Oxford University Press, New York.

[10] Alexander C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fixdahl-King and S. Angel (1977). *A Pattern Language.* Oxford University Press, New York.

[11] Rich C. And R. C. Waters (1990). *The Programmer’s Apprentice*. ACM Press.

[12] Gamma E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

[13] Coplien J. O. and D. C. Schmidt (1995) (eds.) *Pattern Languages of Program Design*. Addison-Wesley.

[14] Vlissides J. M., J. O. Coplien, and N. L. Kerth (1996). *Pattern Languages of Program Design 2*. Addison-Wesley.

[15] Eden Amnon H. and Amiram Yehudai (1997). *Patterns of the Agenda*. LSDF'97: Workshop in conjunction with ECOOP'97. Also available from: `http://www.math.tau.ac.il/~eden/patterns_of_the_agenda.{ps.Z,rtf.zip}`

[16] Kim J. J. and K. M. Benner (1996). “Implementation Patterns for the Observer Pattern”. In [14].

[17] Avotins J., G. Maughan, and C. Mingins (1995). “Language Processor Construction: The Case for YOOCC and TROOPER. *Proceedings of TOOLS USA.*

[18] Eden A. H. and A. Yehudai (1997). *Tricks Generate Patterns*. Technical report 324/97. Also available from: `http://www.math.tau.ac.il/~eden/tricks_generate_patterns.{ps.Z,rtf.zip}`

[19] Liberherr K. J. and I. M. Holland (1989). “Assuring Good Style for Object Oriented Programs”. *IEEE Software*.