Information Systems Frontiers 4:4, 379–391, 2002

A Theory of Object-Oriented Design

Amnon H. Eden
Center for Inquiry, Amherst, NY, USA
E-mail: eden@acm.org

Abstract. *Progress was made in the understanding of object-oriented (O-O) design through the introduction of patterns of design and architecture. Few works, however, offer methods of precise specification for O-O design.*

This article provides a well-defined ontology and an underlying framework for the formal specification of O-O design: (1) We observe key design motifs in O-O design and architectures. (2) We provide a computational model in mathematical logic suitable for the discussion in O-O design. (3) We use our conceptual toolkit to analyze and compare proposed formalisms.

Key Words. *software design theory, software architecture, object oriented programming, formal foundations, design patterns*

1. Introduction

Architectural specifications provide software with "*a unifying or coherent form or structure*" (Perry and Wolf, 1992). Coherence is most effectively achieved through formal manifestations, allowing for unambiguous and verifiable representation of the architectural specifications.

Various formalisms and Architecture Description Languages (ADLs) (Medvidovic and Taylor, 1997) were proposed for this purpose, each of which derives from an established formal theory. For example, Allen and Garlan extend CSP (Hoare, 1985) in WRIGHT (Allen and Garlan, 1997); Dean and Cordy (1995) use typed, directed multigraphs; and Abowd, Allen, and Garlan (1993) chose $\mathbb{Z}$ (Spivey, 1989) as the underlying theory. Other formalisms rely on Statecharts (Harel, 1987) and Petri Nets (Petri, 1962).

In contrast, techniques idiosyncratic to the object-oriented programming (OOP) paradigm, such as *inheritance* and *dynamic binding* (Craig, 1999), induce regularities of a unique nature. Consequently, O-O systems deviate considerably from other systems in their architectures. We would expect the architectural specifications of O-O systems to reflect their idiosyncrasies.

Unfortunately, architectural formalisms largely ignore the O-O idiosyncrasies. Few works (Section 4) recognized the elementary building blocks of *design* and *architecture* patterns. As a result of this oversight, any attempt to use formalisms for the specification of O-O architectures is destined to neglect key regularities in their organization.

Only naturally, coherent specifications warrant the recognition of the underlying abstractions of the paradigm (Odenthal and Quibeldey-Cirkel, 1997). This is equally true for O-O programs. Hence, we observe primitives ("building blocks") in O-O design, such as the *inheritance class hierarchies* (Definition IV) and *clans* (Definition V), which reflect the mechanisms that underlie the paradigm's idiosyncrasies (*inheritance* and *dynamic binding*, respectively.)

1.1. Patterns

More than any other form of documentation, *software patterns* (Coplien and Schmidt, 1995; Vlissides, Coplien, and Kerth, 1996; Martin, Riehle and Buschmann, 1997; Schmidt et al., 2000), in particular *design patterns*, proved effective in capturing recurring motifs in O-O software architecture. Each article of these catalogues addresses a distinct "design pattern", documented in a structured format, and distinguished by a name which carries its intent. Patterns capture abstractions that facilitate communicating problems and solutions. It is safe to argue that the best insight into the regularities of O-O design is provided by the patterns literature.

Yet the genre suffers from a few shortcomings, most prominent appear to be the following:

1. *Ambiguity*. The informal and ultimately fuzzy descriptions puzzle patterns' users and cause

substantial confusion. Even the very pattern writers demonstrate disagreement over their "true meaning" (e.g., pattern discussion[1] and gang-of-four-patterns[2]).

Debates that frequent patterns' users include the following:

- *Instance-of* (Definition III): Whether a particular implementation conforms to a certain pattern;
- *Refinement* (Definition VII): Whether one pattern is a special case of another.

2. *Unstructured knowledge*. With the growth in the number of *patterns* published, the accumulation of information is evolving into an unstructured mass which lacks effective means of indexing.

Clearly, these two problems result from the use of imprecise means of specification, such as verbal descriptions, class diagrams, and concrete examples. This problem can be alleviated by providing formal specifications, which may allow for unambiguous specifications, enable reasoning about the relationships between patterns (e.g., *refinement*), and promote the structuring of the rapidly growing body of patterns.

1.2. Intent

This article introduces the following contributions:

1. Define an ontology that serves as a frame of reference for the discussion in the essential concepts of O-O design. In particular:
   - Observe the building blocks of O-O design, namely, a small set of rudimentary elements that can be used to represent effectively the "design" of many programs;
   - Provide precise definitions for intuitive terms used with reference to O-O patterns, such as *pattern a is a special case of pattern b*.
2. Analyse declarative formalisms for the specification of O-O design patterns. In particular:
   - Reconcile formalisms proposed by translating sample expressions to a unifying framework;
   - Provide criteria for assessing the properties of prospective specification languages (e.g., *expressiveness* and *completeness*).

1.2.1. A terminological note. Our use of the term *pattern* diverges from its use within the patterns community. Inspired by the work of Christopher Alexander (Alexander et al., 1977, Alexander, 1979), a "pattern" is a prescription for solving a category of problems in a specific manner. This prescription is intended for the dissemination of specialized knowledge and to create an instrumental vocabulary.

At the same time, many use "pattern" (particularly with reference to *design patterns*) as a shorthand for a specific segment of the respective article, which focuses primarily is the architectural abstraction manifested in the solution part of the *pattern* (also *microarchitecture*, *lattice* (Eden, 1998), and *leitmotif* (Eden, 2000)). We adhere to the second practice.

1.2.2. A methodological note. We employ symbolic logic as means for precise specification and reasoning. Yet, as in other scientific disciplines where the subject of the statements made (verbal descriptions of patterns) is informal, there is no rigorous way to prove the correctness of some of the statements we make. These claims can only be treated as approximations for some "natural phenomena" (Popper, 1969). As in standard scientific practice, we may only demonstrate evidence that support our hypotheses.

2. Setting the Scene

In this section, we provide a foundation for a formal discourse in O-O design.

2.1. Semantics

Programmers are aware of the conceptual model that accompanies every O-O program: A universe inhabited by classes, operations, attributes, and relations among them. Some of these relations are explicitly expressed as syntactic, built-in constructions of some programming languages, such as *inheritance*, *member-of*, and so forth.

There is a lot to be gained by making this conceptual "stage" into an explicit logic *structure*. Many unsubstantiated claims, such as *this pattern is just a special case of another*, become straightforward facts in this structure.

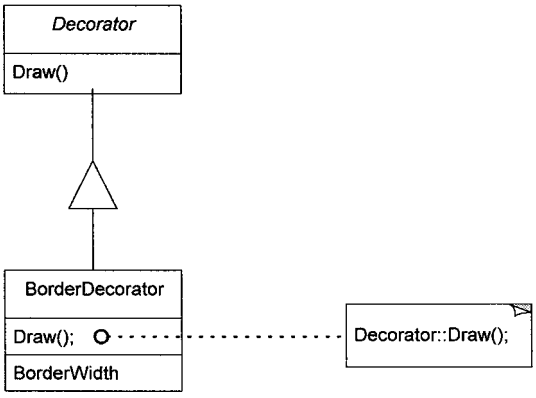
We propose to render this structure explicit exactly like any other mathematical structure, such as algebraic band, boolean algebra, or geometry of points and lines.

The crux of our contribution is, to start with, in recognising the *participants*, their essential *collaborations*, and the suitable way of manifesting them. This practice is not entirely unlike distinguishing the participants in geometric discourse—*points* and *lines*, and the primary relations—*point x is in line y*. Finally, we consider the "staging". In the geometric context, for instance, we ask whether the relation *line x is parallel to line y* is elementary, or whether it is deduced from other relations. Similarly we ask whether *lines* are atomic entities, or perhaps should be represented as sets of *points*, namely, as more elaborated constructs.

The framework we propose incorporates *classes* and *methods* as its atomic entities and relations such as *class d inherit from class b*, *method f is defined in class c*, and *method f creates instances of class c*, as the ground relations amongst them.

In our discussion, a program[3] is abstracted to a simplified representation in mathematical logic called *design model* (Definition 1). Example 1 demonstrates how a simple Java program translates to a *design model*. Extract 1 gives the class diagram of the same program.

In comparison, object notations (e.g., UML (Booch, Jacobson, and Rumbaugh, 1999), OMT (Rumbaugh et al., 1991), Booch (1994), OPEN (Firesmith, Graham, and Henderson-Sellers, 1998)), more specifically *class diagrams*, were used in the specification of design patterns. Class diagrams and other diagrams in object notations, however, are clearly inadequate representation as they only incorporate constant symbols (such as `Decorator` and `Decorator::Draw()` in Extract 1.) Architectural specification, however, specify a *set* of programs indirectly through their properties (Perry and Wolf, 1992). The absence of variable symbols is a one of the major shortcomings of class diagrams. Furthermore, some information about the pattern can only be conveyed using informal "notes".



Extract 1. *OMT diagram of a segment of the* DECORATOR *(Gamma et al., 1994) pattern.*

In the framework we propose, both *methods* and *classes* are represented as "primitive" elements. Hence, the *Design Model* (Definition I) of a program consists only of "ground entities" and of "ground relations", just as a *structure* in mathematical logic.

For example, the association between class `Decorator` and method `Draw` therein is represented using the *DefinedIn* relation as follows:

$$\textit{DefinedIn}(\texttt{Draw},\texttt{Decorator}) \qquad (1)$$

Example 1. Program vs. its model.

```
abstract class Decorator {
  abstract void Draw();
}
class BorderDecorator
  extends Decorator
{
  void Draw() {
     Decorator.Draw(); //...
  }
  int BorderWidth;
}
```

The model of this program consists of the following:

Ground Entities ("participants")

`Decorator`, `BorderDecorator`, `int` of type "class", and `BorderDecorator.Draw`, `Decorator.Draw` of type "method".

Relations ("collaborations")

DefinedIn (`Decorator.Draw`, `Decorator`)
DefinedIn (`BorderDecorator.Draw`, `BorderDecorator`)
Inherit (`BorderDecorator`, `Decorator`)
Reference (`BorderDecorator`, `int`)
Invoke (`BorderDecorator.Draw`, `Decorator.Draw`)
Abstract (`Decorator`)
Abstract (`Decorator.Draw`)
ReturnType (`Decorator.Draw`, `void`)
ReturnType (`BorderDecorator.Draw`, `void`)

Similarly, the relation *Reference* represents the association between class `BorderDecorator` and the type `int` of its attribute as follows:

$$\textit{Reference}(\texttt{BorderDecorator},\texttt{int}) \qquad (2)$$

The simplification of programs into *design models* is an essential step towards the abstraction that is necessary at the architectural level.

In the following definition, the term *object language* refers to the language used for writing source code. For instance, the object language used in Example 1 is Java™ (Gosling, Joy, and Bracha, 2000).

Definition 1 (Design model). Let us denote $\mathcal{L}$ as the *object language*, $\mathbb{T} = \{\mathbb{F}, \mathbb{C}\}$ as a set of *participant types*, and $\mathbb{R} = \{Inherit, Abstract, \ldots\}$ as a set of relation symbols. We assume a mapping function $\mathcal{M}$ such that for each program p in $\mathcal{L}$, $participants_{\mathcal{M}}(p)$ is a set of *ground entities*, each of a type in $\mathbb{T}$, and $collaborations_{\mathcal{M}}(p)$ is a set of n-tuples of entities in $participants_{\mathcal{M}}(p)$, such that each tuple belongs to a relation in $\mathbb{R}$.

We define $\mathfrak{M} = \mathcal{M}(p)$ as the *design model of p according to* $\mathcal{M}$ (in short: "the model of p") as the structure arising from the ground entities in $participants_{\mathcal{M}}(p)$ and the relations in $collaborations_{\mathcal{M}}(p)$.

Further discussion in composition of the sets $\mathbb{T}$ and $\mathbb{R}$ appears in Section 3.

Definition 1 gives rise to a well-defined universe of *design models*, which we denote $\mathcal{P}$. Being abstractions rather than actual programs, an unbounded number of programs are effectively treated as equivalent. Thus, from this point in this article, we disregard "programs" in their original sense and use the term with reference to their abstraction. As we do not refer directly to programs in their original representation, we do not see a difficulty in using this alias.

2.1.1. Static vs. dynamic models. Regardless the choice of object language, static and dynamic conceptual models stand at the heart of the system's design and they exist throughout its lifecycle. Similarly, design patterns determine both structural (namely, static) and behavioural (namely, dynamic) properties.

We focus our discussion on the static model for various reasons. The static components and their correlations are the ingredients of the system's structure and serve as a scaffold that delineates its behaviour. Static properties are far easier to specify, prove or refute, and most importantly, to comprehend. If indeed the primary reason for the continuous software crisis (Gibbs, 1994) is lack of *abstraction*, then there is enough room to improve the structural picture of software systems.

Also, the elements of O-O architecture we observed (listed in Section 3) support the view that, predominantly, design patterns characterize *static* attributes of this universe. Many of the patterns that appear "dynamic" ("behavioural" patterns, as characterized in Gamma et al. (1994)) actually describe a configuration in the static model. By studying leitmotifs of the "behavioural" patterns, one recognizes that, essentially, the behaviour manifested in these abstractions can be expressed through static relations. The examples provided in Section 4 illustrate this observation. Additional examples are provided in Eden (2000).

Finally, OOP is distinguished by *inheritance*, an entirely static abstraction mechanism. This is true not only for statically, strongly typed languages, but also for dynamically typed OOP languages. The graph of inheritance relations is predominant in the characterization of the topology of O-O libraries. Moreover, the organisation of *classes*, *methods*, and their relations, is the "stage" on which the dynamic manifestation of the program takes place, where classes materialise as objects, and methods take the form of messages.

2.2. Syntax

We use higher order predicate calculus in our discussion. Constant names appear in `fixed typeface` and variables in *italics*. We designate the domain of methods by $\mathbb{F}$, and the domain of classes by $\mathbb{C}$. Given a set S, we use $\mathbf{P}(S)$ to denote the power set of S.

Variables should not be confused with constants; thus,

$$\texttt{Draw} \in \mathbb{F} \tag{3}$$

$$\textit{Factories}\colon \mathbf{P}(\mathbb{C}) \tag{4}$$

expression (3) indicates that the symbol `Draw` represents a specific method ("function member"), while (4) declares a variable that ranges over sets of classes.

As established by the "principle of least constraint" (Perry and Wolf, 1992), architectural specifications should not constrain the implementations unnecessarily. This is a desired property of both design patterns and architectural styles, as each specifies a set of desired properties rather that detailing a concrete solution (i.e., a program.)

We conclude that O-O architectural specifications can be expressed as constraints on properties of *participants* and *relations* (Definition 1) and formulated in higher-order logic. This conclusion leads to the following definition:

Definition 2 (Pattern). A *pattern* is defined as a formula $\varphi(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are free variables in φ. We say that $x_1, \ldots, x_n$ are the *participants*, and that the relations in φ are the *collaborations*, that the pattern dictates.

Naturally, Definition 2 is too broad. It offers, however, a baseline for a formal discussion in specifications of patterns in different formal languages. In conjunction with Definition 1, it allows for the formal definition of other useful intuitions, such as an *instance of a pattern* (Definition 3) and *refinement of a pattern* (Definition 7).

2.3. Reasoning

A significant advantage to the use of a formal framework is in the ability to apply rigorous inference rules so as to allow reasoning with the specifications and derive conclusions on their properties.

We can demonstrate simple reasoning by using variables to rephrase the intuitive distinction between the *pattern* π, an *instance* of π (an implementation of π), and a program that contains an instance of π.

The following definition formulates the distinction between a *pattern* and an *instance of a pattern*:

Definition 3 (Instance of a pattern). Let $\varphi(x_1, \ldots, x_n)$ be a pattern. Let $\mathfrak{M}$ designate a model containing the n-tuple of ground entities: $(a_1, \ldots, a_n)$. Let $\mathcal{A}$ be the consistent assignment of $a_1, \ldots, a_n$ to the free variables $x_1, \ldots, x_n$ in φ. If the result of assignment $\mathcal{A}$ in φ is true in $\mathfrak{M}$ then we say that $(a_1, \ldots, a_n)$ *is an instance of* φ *in the context of* $\mathcal{A}$ (also $\mathfrak{M}$ *contains an instance of* φ).

Similarly, we say that a model $\mathfrak{M}$ is *satisfies* φ if there exists some assignment $\mathcal{A}$ such that $\mathfrak{M}$ is an instance of φ in the context of $\mathcal{A}$.

To illustrates Definition 2, consider the following trivial "pattern":

$$\mathit{Invoke}\,(f_1,\ f_2) \tag{5}$$

It is easy to show that the model in Example 1 satisfies (5). To prove this, consider the assignment of `BorderDecorator.Draw` to f_1 and of `Decorator.Draw` to f_2. If we apply this assignment, we get:

$$\mathit{Invoke}\,(\texttt{BorderDecorator.Draw},\ \texttt{Decorator.Draw}) \tag{6}$$

which is true in Example 1.

Corollary 1. *From Definition* 1 *and Definition* 2 *we conclude that each* pattern *(regardless the specification language used) specifies a subset of* $\mathcal{P}$.

3. The Building Blocks of O-O Architecture

In this section, we observe the elements of O-O architecture, which we refer to as *rudiments*, and illustrated them using examples from (Gamma et al., 1994). The discussion is broken down along the distinction between *participants* and *collaborations* as the elements of patterns (Gamma et al., 1994).

3.1. Participants

This section is dedicated to abstractions that represent elementary and composite *participants*.

3.1.1. Rudiment A: Ground entities. The predominant agents in O-O software, as well as the *participants* of every single pattern in Gamma et al. (1994), are *classes, methods*, and *objects*. Since we focus in static specifications, we will only discuss the first two. Formula (7) illustrates a declaration of the participants in Example 1:

$$\begin{aligned}&\texttt{Decorator},\texttt{BorderDecorator},\texttt{int} \in \mathbb{C}\\ &\texttt{Decorator.Draw},\\ &\quad\texttt{BorderDecorator.Draw} \in \mathbb{F}\end{aligned} \tag{7}$$

We regard classes and methods as atomic ("primitive") elements, or *ground entities*. Their properties, e.g., the arguments of a method, data members of a class, and so forth, are expressed through *relations* (Rudiment E).

3.1.2. Rudiment B: Higher-order entities. Most often, participants appear in unbounded sets consisting of elements of a uniform type, namely, either classes or methods. Uniform sets of participants playing a specific

role (e.g., "creators", "visitors", "products", etc.) are omnipresent in design patterns. Numerous examples appear in every pattern.

Higher order sets (i.e., uniform sets of uniform sets) are also omnipresent. Higher order sets are also uniform in the sense that their elements are of uniform type and order. Consider, for example, the participants in the *ABSTRACT FACTORY* pattern: *products* is a set of sets of classes, while *factory methods* is a set of sets of methods. Similarly, observe the set of sets of $visit(element_i)$ methods of the *VISITOR* pattern. Formally:

Definition 4 (Higher-dimension entity).

(i) A ground entity is said to have *dimension* 0;
(ii) A set that comprises classes (methods) of dimension $d-1$ is called a *a class (method) of dimension d*;

For example, we can refer to the set of *Visitor* classes as *a class of dimension 1*, and to the set of *Visit* methods as *a method of dimension 2*.

3.1.3. Rudiment C: Clans. Dynamic binding, the mechanism for the dynamic selection of a method, is fundamental to OOP. It is enabled by a combination of inheritance and shared signatures. A *clan* is the static manifestation of the "family" of methods that share a particular dispatch table. Formally:

Definition 5 (Clan). W say that method F is *a clan in class C* iff the following conditions hold:

(i) All methods in F share the same signature;
(ii) Each method in F is defined in a different class in C.

A formulation of Definition 5 in predicate calculus, which uses the relations *SameSignature* and *DefinedIn* can be found in Eden (2000). Observe that a clan is always defined with relation to a given set of classes, not as an isolated property.

Clans are ubiquitous and occur wherever dynamic binding is used. The following are examples of clans: the set of $visit(element_i)$ methods of the *VISITOR* pattern; the set of $create(product_i)$ of the *ABSTRACT FACTORY*; the set of the *update* methods of the *OBSERVER* pattern; the set of *ConcreteAlgorithms* of the *STRATEGY* pattern; and so forth. In fact, every time inheritance is used there is a very high chance that dynamic binding is also present.

Observe another common construction: A set of clans T such that every clan $F \in T$ is defined in class C. This abstraction is termed *tribe* and it can be found in almost every pattern (Eden, 2000).

3.1.4. Rudiment D: Hierarchies. *Inheritance* is also fundamental to OOP, and inheritance class hierarchies occur in every O-O program. Most dominant, however, is a particular construction that consists of a single inheritance hierarchy. A 1-dimensional class is termed *hierarchy* if it contains an abstract ("root") class from which all other ground classes inherit (possibly indirectly).

In the following definition, $\mathcal{R}^+$ designates the transitive closure of (one or more) occurrences of the relation $\mathcal{R}$.

Definition 6 (Hierarchy). A 1-dimension class $\mathtt{h}$ is a *hierarchy* if the following condition holds:

$$\exists\, root \in \mathtt{h} \bullet Abstract(root) \wedge \forall\, cls \in \mathtt{h} \bullet Inherit^{+}(cls, root)) \qquad (8)$$

Observe also the occurrences of *sets of hierarchies*, e.g., the set of *product* hierarchies in the *ABSTRACT FACTORY* (Extract 4).

Almost every pattern in Gamma et al. (1994) contains one or more *hierarchies*. For example: The set of *concrete-observers* with the abstract *observer* (*OBSERVER* pattern); the set of *concrete-products* with the abstract *product* (*FACTORY METHOD* pattern); and the set *concrete-commands* with the abstract *Command* (*COMMAND* pattern). Henceforth, the term *hierarchy* is used only in the sense defined above.

3.2. Collaborations

In this section, we observe key regularities in the correlations among the patterns' *participants*.

3.2.1. Rudiment E: Ground Relations. We identify a small group of ground relations which, as illustrated in Eden, Hirshfeld, and Yehudai (1999), is sufficient to account for most types of collaborations that occur among ground entities in the GoF patterns. Principal relations are listed in Table 1. Example 1 illustrated how ground relations model correlations among the elements of a program.

Table 1. *Intuitive interpretations for ground relations*

Abstract: $\mathbb{F} \cup \mathbb{C}$
Indicates whether the entity is abstract (In Eiffel: *deferred*)

Assign: $\mathbb{F} \times \mathbb{C} \times \mathbb{C}$
Indicates that a reference from one class to the other is assigned within the body of a given method.

Create: $\mathbb{F} \times \mathbb{C}$
Indicates that the body of the method contains an expression whose evaluation creates an instance of the class. Such expressions include, for example:

```
new int number;          (Java)
int numbers[2];          (C++)
if (string("A") == s)    (C++)
!INTEGER! num.make;      (Eiffel)
```

DefinedIn: $\mathbb{F} \times \mathbb{C}$
Indicates that the method is a member in the class

Forward: $\mathbb{F} \times \mathbb{F}$
Forward (*f, g*) means that f *invokes* g with the additional requirement that the actual arguments in the invocation expression are the formal arguments of f. Such a method in C++ will look as follows:

```
void TCPConnection::ActiveOpen(int t)
  {_state->ActiveOpen(t); }
```

Invoke: $\mathbb{F} \times \mathbb{F}$
Invoke (*f*, *g*) indicates that f "may invoke" g, namely, that within the body of method f there is an expression whose evaluation invokes g. *Note that we ignore the control flow*

Inherit: $\mathbb{C} \times \mathbb{C}$
Indicates that the class on the left inherits from the class on the right

Reference [*-To-Many*]: $\mathbb{C} \times \mathbb{C}$
Indicates that class on the left defines an attribute [multiple attributes] whose type is specified by the class on the right

ReturnType: $\mathbb{F} \times \mathbb{C}$
Indicates the return type of a method

SameSignature: $\mathbb{F} \times \mathbb{F}$
Indicates that the two functions have the same name and formal arguments.

3.2.2. Rudiment F: Bijections. Consider the following quote from the specification of the *PROXY* (Gamma et al., 1994, p. 270): "*each proxy operation ... forwards the request to the subject.*" It implies that for every method p in class *Proxy* there is exactly one method s in class *Subject* such that *Forward* (p, s). In other words, the relation *Forward* is a bijective function between the sets.

It is particularly interesting to observe bijections in higher-dimensional entities. Consider, for example, the relation which occurs in the *ABSTRACT FACTORY* between the set of clans *create* $(product_i)$ (namely, a clan for each product), denoted *FactoryMethods*, and the set of "product" hierarchies, which we denote *Products*. Their collaboration is described as follows: "*AbstractFactory ... defines a different operation for each kind of product it can produce*." (Gamma et al., 1994, p. 87) This description implies that *Create* is a bijection between *FactoryMethods*, which is a 2-dimensional method, and *Products*, a 2-dimensional class.

Bijections exist in almost every pattern of the GoF catalogue, as demonstrated in Eden (2000).

4. Specification Languages

Mathematical logic provides numerous formalisms for deliberating algebraic constructs such as the one proposed in Definition I, from first order predicate calculus to higher order languages, which afford the representation of higher order sets, functions, and relations. Different publications present different views on the suitable formalism for the representation of patterns. Below we discuss a selection of these languages.

Observe that the publications discussed provide one or two examples each, and, with the exception of LEPUS, do not provide a complete semantic specification. This greatly limits the scope of our analysis. Thus, this section focuses on examples rather than complete definitions. We hope more comprehensive results can be obtained in the future.

4.1. DisCo

Defined as an extension of *temporal logic of actions* (Lamport, 1994), DisCo (Mikkonen, 1998) consists of constructions that express the composition of classes and the semantics of methods. Although the temporal logic-based formalism focuses on dynamic specifications, many specifications clearly map to our static framework. Consider for example Extract 2, the

class *Subject*={*Data*}
class *Observer*={*Data*}
relation (0..1)·*Attached*·(*):
  Subject × *Observer*
Attach (*s*:*Subject*; *o*:*Observer*):
  ¬ *s*·*Attached*·*o*
  → *s*· *Attached*·*o*

Extract 2. *OBSERVER in DisCo (Mikkonen, 1998).*

specification of the *OBSERVER* pattern (Gamma et al., 1994). It describes the relationships between three classes and defines the semantics of one method in terms of pre- and postconditions. Despite the seemingly "dynamic" nature of this specification, formula (9) demonstrates how we can rephrase it in terms of static relations.

$$\begin{array}{l} Subject, Observer, Data : \mathbb{C} \\ Attach : \mathbb{F} \end{array} \qquad (9)$$

$$\begin{array}{l} Reference\,(Subject, Data) \\ Reference\,(Observer, Data) \\ Assign\,(Attach, Subject, Observer) \end{array}$$

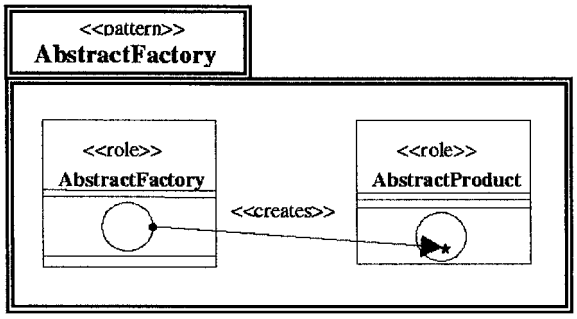
4.2. Constraint diagrams

Described as a precise visual specifications language combined with elements in UML (Booch, Jacobson, and Rumbaugh, 1999), constraint diagrams (Lauder and Kent, 1998) are a visual representation of first order set-theoretic relations. Extract 3 depicts the "role model" of the *ABSTRACT FACTORY* pattern.

Rather than repeating here the interpretation of Extract 3, we both explain it and demonstrate how it translates to our static framework in a single formula.

$$AbsFactory, AbsProduct : \mathbb{C} \qquad (10)$$

$$\begin{array}{l} \forall\ ConcFactory, ConcProduct \bullet \\ \quad Inherit^{+}(ConcFactory, AbsFactory) \wedge \\ \quad Inherit^{+}(ConcProduct, AbsProduct) \Rightarrow \\ \exists\ fm : \mathbb{F} \bullet \\ \quad DefinedIn\,(fm, ConcFactory) \wedge \\ \quad Create\,(fm, ConcProduct) \end{array}$$



Extract 3. *"Role Diagram" of the ABSTRACT FACTORY.*

Factories : $\mathbb{H}$ *Products* : $\mathbf{P}(\mathbb{H})$ *FMs* : $\mathbf{P}(\mathbb{S})$
$Create^{\leftrightarrow}$ *(FMs⊗Factories,Products)* $ReturnType^{\leftrightarrow}$ *(FMs⊗Factories,Products)*

Extract 4. *ABSTRACT FACTORY in* LEPUS.

4.3. LEPUS

LEPUS (Eden, 2000) is a language for the specification of O-O software architecture. Consider for example the symbolic specification of the *ABSTRACT FACTORY* in LEPUS, which appears in Extract 4. The interpretation of Extract 4 is as follows:

1. $\mathbb{H}$ is the domain of hierarchies, so that *Factories* is a hierarchy variable and *Products* ranges over sets of hierarchies.
2. $\mathbb{S}$ is a domain of all "signatures" of methods. Thus, *FMs* is a variable which represents the set of signatures of the factory methods.
3. The expression $FMs \otimes Factories$ incorporates the *selection operator* $\otimes$ and yields all methods defined in (a class in) *Factories* whose signature is in *FMs*, namely, the set of factory-method clans.
4. The abbreviation $\mathcal{R}^{\leftrightarrow}(V, W)$ represents a *bijection* (Rudiment F) between the sets V, W. Thus, the two statements below the dividing line indicate that for every factory method *fm* there is a product p such that is *Create*(*fm,p*) and *ReturnType*(*fm,p*).

These examples demonstrate how the framework we propose sheds light on the difference between various formalisms.

In addition to the results analysed above, following is a short overview of other results that were reported in literature.

4.4. Design patterns' formalisms

Formal Languages for the specification of design patterns are described in several publications. Helm, Holland and Gangopadhyay (1990) defined "Contracts", an extension to first order logic with representations for function calls, assignments, and an ordering relation between them. The "behavioural compositions" described do not address structural relations but only behavioural characterizations.

4.5. Tool support

Florijn, Meijers, and van Winsen (1997) propose a tool that supports the application of design patterns. They use the "Fragments Model" for the representation of patterns, a graph with labelled arcs, whose nodes stand for the *participants* in the pattern, and its arcs describe the roles of the connecting nodes. Predominantly, the *patterns' wizard* (Eden, Gil, and Yehudai, 1997) supports the *application* of patterns through their representation as metaprogramming algorithms, namely, by the sequence of steps in their application. Other tools supporting the application of design patterns are described in Pal (1995), Budinsky et al. (1996), Quintessoft Engineering (1997), Brown (1996), Alencar et al. (1999), Kramer and Prechelt (1996), Bosch (1996), and O'Cinnéide and Nixon (1999), most of which are reviewed in Eden, Hirshfeld, and Yehudai (1999).

4.6. Ground relations

Other works have observed "elementary" relations between entities. Keller et al. (1999) list *Call Actions* and *Create Actions* (corresponding to *Invoke* and *Create* relations of Table 1, respectively) among the information their reverse engineering environment maintains for the representation of O-O programs, as well as information on the relations between classes (e.g., *Inheritance*, *Reference*). Chiba (1995) describes a metaprogramming environment for manipulating C++ programs whose features are classified under *Function Invocation* and *Object Creation*, among other features.

5. Relations Among Patterns

This section discusses relations among patterns as suggested by informal means and offers precise definitions for these terms by means of the framework defined so far.

5.1. Refinement

Two articles (Agerbo and Cornils, 1998; Vlissides, 1997), describe one pattern as a "refinement" of another, meaning that one is a special case of the other. Cargill (1996) describes "categories of patterns," where one category "refines" the other. Kim and Benner (1996) describe the *push* and *pull models* as "refinements of the *OBSERVER* pattern" (Gamma et al., 1994). Similarly, Rohnert (1996) describes "specializations of the *PROXY* pattern," such as *CACHE PROXY* and *PROTECTION PROXY*. The following definition formalises this intuition:

Definition 7 (Refinement). We say that pattern ρ *refines* pattern π, written as: $\rho \models \pi$, iff for every assignment $\mathcal{A}$ that satisfies ρ in some model $\mathfrak{M}$, $\mathcal{A}$ also satisfies π in $\mathfrak{M}$.

By Definition 7, *refinement* is equivalent to *semantic entailment* (Huth and Ryan, 2000). Thus, we may conclude that whenever our specification language allows for a proof theory and a *complete* and *sound* relation $\vdash_{\mathcal{L}}$, then the refinement relation $\rho \models \pi$ holds if and only if $\rho \vdash_{\mathcal{L}} \pi$ holds.

The difficulty in handling *refinement* in the lack of a formal theory is well demonstrated in a classic example (Vlissides, 1997). The article reports a debate between the authors of the *OBSERVER* pattern (Gamma et al., 1994), which could not agree whether it is indeed refined by the *MULTICAST* (a pattern proposed by J. Vlissides). Eden, Gil, Hirshfeld, and Yehudai (1998), show how the debate can be resolved using LEPUS (Section 4).

5.2. Projection

In simple terms, *projection* is obtained by replacing a uniform set of participants X with a single participant x of the same type. Formally:

Definition 8 (Projection). A *projection of the free variable* $X : \mathbf{P}(\mathbb{T})$ *in* $\varphi(X)$ is a formula $\varphi(x)$ resulting in the consistent replacement of X with $x : \mathbb{T}$ in φ.

We say that $\varphi(X)$ is an *abstraction* of x in $\varphi(x)$.

Example 2 illustrates this definition:

Example 2 (Projection). Extract 4 gives the definition of the *ABSTRACT FACTORY* pattern in LEPUS. The specification of the *FACTORY METHOD* is obtained merely by modifying the dimension of two variables in the *ABSTRACT FACTORY* as follows:

ABSTRACT FACTORY	*FACTORY METHOD*	*Description*
Products: $\mathbf{P}(\mathbb{H})$	*Products*: $\mathbb{H}$	A hierarchy vs. a set of hierarchies
FMs: $\mathbf{P}(\mathbb{S})$	*FMs*: $\mathbb{S}$	A signature vs. a set of signatures

6. Comparative Criteria

With the increase in the popularity of design patterns, we expect that even more formalisms should arise. The purpose of this section is to define properties that can be used as criteria in comparing between such formalisms.

We maintain that shorter expressions contribute to a clearer language. The *concision* criterion allows us to judge the relative "shortness" of expressions.

In the following definition, $\mathcal{L}$, $\mathcal{L}_1$ and $\mathcal{L}_2$ designate specification languages, Π is a set of patterns, φ_π is the expression of pattern π in $\mathcal{L}$, and c is a numeric constant.

Definition 9 (Concision). Let us assume a metric function that measures the length of expressions in $\mathcal{L}_1$ and $\mathcal{L}_2$:

$$Len : \mathcal{L}_1 \cup \mathcal{L}_2 \to \mathbb{N}$$

Let φ_1 and φ_2 be specifications in $\mathcal{L}_1$, $\mathcal{L}_2$, respectively. We say that φ_1 is more concise than φ_2 iff

$$Len(\varphi_1) < Len(\varphi_2)$$

We say that $\mathcal{L}$ is *c-concise with respect to* Π iff for every π in Π, the following is true:

$$Len(\varphi_\pi) < c$$

Naturally, different formalisms give rise to different subsets of $\mathcal{P}$, and thus may or may not account for subsets of interest. Yet the question whether a certain formula expresses an informal description is difficult to answer conclusively exactly because the contemporary means of specification are ambiguous. As a step towards measuring their capacity we define *expressiveness* by means of the rudiments made in Section 3:

Definition 10 (Expressiveness). We say that a specification language $\mathcal{L}$ is *expressive* if it incorporates the rudiments listed in Section 3, namely:

- participants of any dimension (Definition 4)
- ground and set relations (Rudiment E and Rudiment F)
- clans (Definition 5)
- hierarchies (Definition 6)

Observe that, while different formalisms may interpret 'class' and 'method' differently, Definition 10 does not depend on these variations. And justly so, as architectural specifications need not be affected by these variations.

7. Future Directions

We observe several directions of future research:

7.1. Compactness

Definition 10 is satisfied by any language that assimilates the *rudiments* of Section 3, such as higher order logic.[4] Observe, however, the risk of getting a specification language that is "too expressive" or too large, meaning that it incorporates many more expressions than necessary.[5] For example, an architectural specification language is not expected to account for the following sets of programs:

- The set of programs that guarantee *liveness*
- The set of programs that can be written in the Java™ programming language
- The set of programs that the do not terminate

A "leaner" language is better because of the following reasons, among others:

1. *Clarity.* Excessive number of possible expressions increases the "complexity" of the specification language and makes it difficult for its users to understand it and to use it.
2. *Reasoning.* The properties of "simpler" and smaller languages can be reasoned upon more easily.
3. *Discovery.* Semi-automation of a "discovery" process, namely, the search for "new patterns" in programs, may become more feasible by reducing the set of candidate patterns. By restricting the specification language and constraining its search, a tool is more likely to create meaningful results.
4. *Abstraction.* As abstractions, patterns depict only essentials and eliminate irrelevant details. Elimination of details is expected to lead to a smaller language.

To summarize, *compactness* has the intuitive meaning of the property which characterizes languages

with fewer "unreasonable" patterns. It would be useful to convey the idea of "unreasonable" patterns by providing a measurement for *compactness* at this point.

Dynamic specifications. Despite the achievement in specification through static properties, our framework does not allow the expression of certain behavioural conditions. Formalisms that may be of use are Temporal Logic of Actions (Lamport, 1994), as in Mikkonen (1998), and Abstract State Machines (Gurevich, 1994).

Applying the criteria. Once complete definitions and a sufficient number of examples are provided for proposed formalisms, our analysis can be improved in many ways. An interesting result can be achieved by applying the *criteria* suggested so as to compare sample specifications.

8. Conclusions and Summary

We present a formal framework for the discussion in O-O software architecture by observing elementary and composite abstractions in its specification and by offering a logic model for representing and deliberating these abstractions. We have demonstrated how examples in various pattern specification languages map to our framework. We have used our framework to define informal concepts in formal terms. Finally, we offered means for comparing the properties of architectural formalism.

Acknowledgments

This work would not have been possible if not for the significant contributions of Yoram Hirshfeld of Tel Aviv University. Special thanks to Björn Victor of Uppsala University for his insights and detailed comments. Many thanks to Peter Grogono for his feedback.

Notes

1. `pattern-discussion`. Mailing list: `http://hillside.net/patterns/Listss.html`
2. `gang-of-four-patterns`. Mailing list: `http://hillside.net/`
3. For the purpose of this article, "program" is any text that is considered well-formed by the rules of the respective programming language.
4. A construction of these elements in higher order logic appears in Eden, Hirshfeld, and Yehudai (1999).
5. UML is an example that comes to mind.

References

Abowd G, Allen R, Garlan D. Using style to understand descriptions of software architecture. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1993.

Agerbo E, Cornils A. How to preserve the benefits of design patterns. In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 18–22, Oct., Vancouver, Canada, 1998:134–143.

Alencar P, Cowan D, Dong J, Lucena C. A pattern-based approach to structural design composition. *IEEE 23rd Annual International Computer Software and Application Conference*, 1999:160–165.

Alexander C. *The Timeless Way of Building*. New York: Oxford University Press, 1979.

Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fixdahl-King I, Angel S. *A Pattern Language*. New York: Oxford University Press, 1977.

Allen R, Garlan D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology* 1997;6(3):213–249.

Booch G. *Object Oriented Analysis and Design with Applications, 2nd edn*. Benjamin/Cummings, 1994.

Booch G, Jacobson I, Rumbaugh J. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Bosch J. Relations as object model components. *Journal of Programming Languages*, 1996;4(1):39–61.

Brown K. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk* M. Sc. Thesis, University of Illinois, 1996

Budinsky FJ, Finnie MA, Vlissides. JM, Yu PS. Automatic code generation from design patterns. *IBM Systems Journal*, 1996 35(2):151–171.

Cargill T. Localized ownership: Managing dynamic objects in C++. In: Vlissides JM, Coplien JO, Kerth NL, eds. *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley, 1996.

Chiba S. A metaobject protocol for C++. In: *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 15–19, Oct., Austin, USA 1995:285–299.

Coplien J, Schmidt D, eds. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

Craig I. *The Interpretation of Object-Oriented Programming Languages*. Berlin: Springer-Verlag, 1999.

Dean TR, Cordy JR. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering*, 1995; 21(4).

Eden AH. Giving 'The quality' a name. *Journal of Object Oriented Programming* 1998;11(3):5–11.

Eden AH. *Precise specification of design patterns and tool support in their application*. Ph.D. Dissentation, Department of Computer Science, Tel Aviv University, 2000.

Eden AH, Gil J, Yehudai A. Precise specification and automatic application of design patterns. In: *Proceedings of the Twelve IEEE International Automated Software Engineering Conference (ASE 1997), 3–5, Nov, Lake Tahoe, Nevada*, Los Alamos: IEEE Computer Society Press, 1997:143–152.

Eden AH, Hirshfeld Y, Yehudai A. Multicast–Observer ≠ typed message." C++ *Report*, 1998;10(9):33–39.

Eden AH, Hirshfeld Y, Yehudai A. *Towards a mathematical foundation for design patterns.* Technical Report 1999-004, Department of Information Technology, Uppsala University, 1999.

Firesmith DG, Graham I, Henderson-Sellers B. *Open Modeling Language (Olm) Reference Manual*. Cambridge University Press, 1998.

Florijn G, Meijers M, van Winsen P. Tool support in design patterns. In: Askit M, Matsuoka S, eds. *Proceedings of the 11th European Conference on Object Oriented Programming—ECOOP 97*, Lecture Notes in Computer Science, vol. 1241. Berlin: Springer-Verlag, 1997.

Gamma E, Helm R, Johonson R, Vlissides J. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley, 1994.

Gibbs WW. Software's chronic crisis. *Scientific American* 1994;86–95.

Gosling J, Joy B, Bracha G. *The Java™ Language Specification, 2nd edn.* Reading, MA: Addison Wesley Longman, 2000.

Gurevich Y. Evolving algebras. In: Pehrson B, Simon I, eds. *IFIP 13th World Computer Congress 1994*, vol. I: Technology and Foundations, 1994:423–427.

Harel D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 1987;8(3):231–274.

Helm R, Holland IM, Gangopadhyay D. Contracts: Specifying behavioral compositions in object-oriented systems. In: *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, 21–25, Oct., Ottawa, Canada, 1990:169–180.

Hoare CAR. *Communicating Sequential Processes*. New Jersey: Prentice-Hall, 1985.

Huth MR, Ryan MD. *Logic in Computer Science: Modeling and Reasoning About Systems*. Cambridge, UK: Cambridge University Press, 2000.

Keller RK, Schauer R, Robitaille S, Page P. Pattern-based reverse engineering of design components. In: *Proceedings of the Twenty-First International Conference on Software Engineering*, Los Angeles, CA: IEEE, 1999:226–235.

Kim JJ, Benner KM. Implementation patterns for the observer pattern. In: Vlissides J, Coplien JO, Kerth NL, eds. *Pattern Languages in Program Design 2*. Reading, MA:Addison-Wesley, 1996.

Kramer C, Prechelt L. Design recovery by automated search for structural design patterns in object-oriented software. In: *Proceedings of the Working Conference on Reverse Engineering,* 8–10, Nov., Monterey, CA, 1996:208–215.

Lamport L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 1994;16(3):872–923.

Lauder A, Kent S. Precise visual specification of design patterns. In: *Proceedings of the 12th European Conference on Object Oriented Programming*, Brussels, Belgium, Lecture Notes in Computer Science, vol. 1445, Jul, Eric, ed. Berlin: Springer-Verlag, 1998.

Martin R, Riehle D, Buschmann F, eds. *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1997.

Medvidovic N, Taylor RN. A framework for classifying and comparing architecture description languages. In: *Proceedings of the Sixth European Software Engineering Conference*, with *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 22–25, September, Zurich, Switzerland, 1997:60–76.

Mikkonen T. Formalizing design patterns. In: *Proceedings of the International Conference on Software Engineering*, 19–25, April, Kyoto, Japan, 1998:115–124.

O'Cinnéide M, Nixon P. A methodology for the automated introduction of design patterns. In: *Proceedings of the IEEE International Conference on Software Maintenance*, 30 August–3 Sept., 1999.

Odenthal G, Quibeldey-Cirkel K. Using patterns for design and documentation. In: *Proceedings of the European Conference of Object Oriented Programming*, 1997, Lecture Notes in Computer Science. Berlin: Springer, 1997.

Pal PP. Law-governed support for realizing design patterns. In: *Proceedings of 17th Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17)*, Englewood Cliffs, NJ: Prentice Hall, 1995.

Perry DE, Wolf AL. Foundation for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 1992;17(4):40–52.

Petri CA. *Communications with automata*. Technical Report RADC-TR-65-377, Applied Data Research, Princeton, NJ, 1962.

Popper K. *Conjectures and Refutations.* London: Rutledge, 1969.

Quintessoft Engineering, Inc. C++ Code Navigator 1.1. `http://www.quintessoft.com`, 1997.

Rohnert H. The proxy design pattern revisited. In: Vlissides J, Coplien JO, Kerth NL, eds. *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley, 1996.

Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

Schmidt D, Stal M, Rohnert H, Buschmann F. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, 2000.

Spivey JM. *The Z Notation: A Reference Manual*. New Jersey: Prentice-Hall, 1989.

Vlissides J. Multicast. C++ *Report*, vol. 9, no. 8. New York, NY: SIGS Publications, 1997.

Vlissides JM, Coplien JO, Kerth NL. *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley, 1996.

Amnon H. Eden graduated the 4th class of the *inter-disciplinary programme of excellence* in Tel Aviv University and received his Ph.D. in 2000. He worked as a consultant with software firms and chaired the *software engineering transition programme* at the Tel Aviv College of Management. Presently, Dr. Eden is with the Department of Computer Science at the University of Essex, UK, and a research scholar at the *Center for Inquiry* in Amherst, NY.