

PRINCIPLES IN FORMAL SPECIFICATION OF OBJECT ORIENTED DESIGN AND ARCHITECTURE

Amnon H. Edenⁱ

*Department of Computer Science
Concordia University
Montreal, Canada H3G 1M8*

Yoram Hirshfeld

*Department of Mathematics
Tel-Aviv University
Tel Aviv 69978, Israel*

Abstract

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

This article provides a well-defined ontology and an underlying framework for the formal specification of O-O architectures: We observe key regularities and elementary design motifs in O-O design and architectures; define “architectural model” in logic; and formulate relations between specifications. We demonstrate how to declare and reason with the representations. Finally, we use our conceptual toolkit to compare and evaluate proposed formalisms.

Keywords: Software architecture, object oriented programming, formal foundations, software patterns

1. Introduction

Architectural specifications provide software with “a unifying or coherent form or structure” [1]. 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) [2] were proposed for this

purpose, each of which derives from an established formal theory. For example, Allen and Garlan extend CSP [3] in *Wright* [4]; Dean and Cordy [5] use typed, directed multigraphs; and Abowd, Allen, and Garlan [6] chose \mathbb{Z} [7] as the underlying theory. Other formalisms rely on Statecharts [8] and Petri Nets [9].

In contrast, techniques idiosyncratic to the object-oriented programming (OOP) paradigm, such as *inheritance* and *dynamic binding* [10], 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 have largely ignored the O-O idiosyncrasies. Few works (reviewed in 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 [11]. This is equally true for O-O programs. Therefore, we observe a small set of primitives (“building blocks”) that are proved sufficient to express O-O architecture. These include abstractions such as the *inheritance class hierarchies* (Definition VI) and *clans* (Definition V), which are induced by the mechanisms of *inheritance* and *dynamic binding*, respectively. Having

ⁱ Email: eden@acm.org, phone: +1 (514) 848 3073, fax: +1 (514) 848 2830

observed the underlying abstractions we can define them formally and use them to defined the more elaborated constructions.

Patterns

More than any other form of documentation, *software patterns* [12, 13, 14, 15], in particular *design patterns*, were successful in capturing recurring motifs in O-O software architecture. Each pattern addresses a separate configuration of design or architecture, documented in a structured format, and distinguished by a name which carries its intent. Patterns carry abstractions that facilitate communicating problems and solutions.

Although the best insight into the regularities of O-O architecture is provided by the patterns literature, it appears that the genre has not matured as a structured engineering domain. In particular, we observe two problems in this literature:

1. **Ambiguity.** The informal and ultimately fuzzy descriptions puzzle pattern users and cause substantial confusion. Even the very pattern writers demonstrate disagreement over their “true meaning” (e.g., [16, 17]).

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 any 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 architecture. In particular:
 - ◆ Observe the building blocks of O-O architecture, namely, a small set of elementary ingredients that is sufficient to express many architectures;
 - ◆ Provide precise definitions for intuitive terms used with reference to O-O patterns;
 - ◆ Establish relations between patterns which were of mere intuition or in dispute;
2. Analyse declarative formalisms for the specification of O-O design patterns. In particular:
 - ◆ Reconcile formalisms proposed by translating sample expressions to our framework;
 - ◆ Provide criteria for assessing the properties of prospective specification languages (e.g., *expressiveness* and *completeness*).

A Terminological Note

Our use of the term *pattern* diverges from its use within the patterns community. Inspired by the work of Christopher Alexander [18, 19], 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 this prescription, which is the architectural abstraction manifested in the solution part of the *pattern* (also *microarchitecture*, *lattice* [20], and *leitmotif* [21]). Since we concern ourselves with software architectures, we adhere to the second practice.

A Methodological Note

We employ symbolic logic as means for precise specification and reasoning. Yet, as in other scien-

tific disciplines where the subject of the statements made (verbal descriptions of patterns in our case) 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” [22]. As in standard scientific practice [22], we seek to convince the reader by evidence that support our hypotheses.

2. Setting the Scene

In this section, we provide a foundation for a formal discourse in static O-O architecture and patterns.

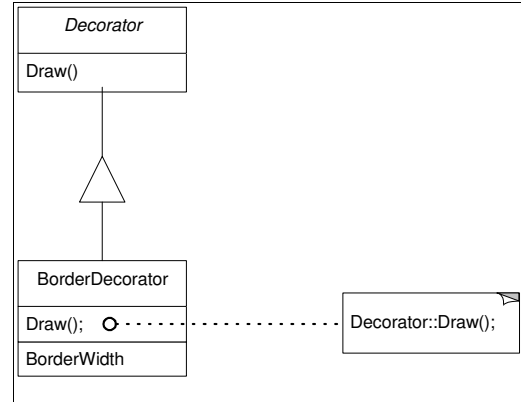
2.1 Semantics

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 “primary” (i.e., 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, similar to *points* and *lines*, and the relations *class d inher-*



Extract 1: OMT diagram of a segment of the *DECORATOR* [26] pattern

its from class *b*, method *f* is defined in class *c*, and method *f* creates instances of class *c*, as the ground relations amongst them.

Thus, every program (i) is represented by a simplified logic representation called *architectural model* (Definition I). Example 1 demonstrates how a simple Java program translates to a *model*. Extract 1 gives the class diagram of the same program.

Object notations (e.g., UML [23], OMT [27], Booch [24], OPEN [25]), 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 class `Decortr`, method `Decortr::Draw`, and so forth.) Architectural specification, as illustrated by Perry and Wolf [1], require generic expressions which set forth abstract constraints, thereby specifying a *set* of programs indirectly through their properties (AKA *intentional specifications*.) The absence of variable symbols is a one of the major shortcomings of object diagrams.

For example, we would like to express the fact that `BorderDecorator` may inherit indirectly (that is, through intermediate classes) from `Decorator`. Observe that, in fact, the class diagram in Extract 1 is roughly equivalent to the Java™ program in Example 1.

Furthermore, observe that some information about the pattern that is specified in OMT [27]

i For the purpose of this article, “program” is any text that is considered well-formed by the rules of the respective programming language.

class diagrams, such as the one depicted in Extract 1, can only be conveyed using informal “notes”.

In addition, the standard practice in object notations is that *methods* (also *function members*, *routines*) and *attributes* (also *data members*, *instance variables*) are represented as elements of the respective class. Seeking a simpler and more manageable representation, we represent both *methods* and *classes* as “primitive” elements. Hence, the *architectural model* (Definition I) of a program consists only of “ground entities” and of “ground relations”, just as logic model.

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

$$\text{DefinedIn}(\text{Draw}, \text{Decorator}) \quad (1)$$

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

$$\text{Reference}(\text{BorderDecorator}, \text{int}) \quad (2)$$

The simplification of programs into architectural 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 programming language that was used to write source code. For instance, the object language used in Example 1 is Java™ [28].

Definition I: Architectural Model

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

We define $\mathbb{M} = \mathcal{M}(p)$ as the *architectural model of p according to \mathcal{M}* (in short: “the model of p ”) as the model arising from the

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”):
 $\text{DefinedIn}(\text{Decorator}::\text{Draw}, \text{Decorator})$
 $\text{DefinedIn}(\text{BorderDecorator}::\text{Draw}, \text{BorderDecorator})$
 $\text{Inherit}(\text{BorderDecorator}, \text{Decorator})$
 $\text{Reference}(\text{BorderDecorator}, \text{int})$
 $\text{Invoke}(\text{BorderDecorator}::\text{Draw}, \text{Decorator}::\text{Draw})$
 $\text{Abstract}(\text{Decorator})$
 $\text{Abstract}(\text{Decorator}::\text{Draw})$
 $\text{ReturnType}(\text{Decorator}::\text{Draw}, \text{void})$
 $\text{ReturnType}(\text{BorderDecorator}::\text{Draw}, \text{void})$

ground entities in $\text{participants}_{\mathcal{M}}(p)$ and the relations in $\text{collaborations}_{\mathcal{M}}(p)$.

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

Definition I gives rise to a well-defined universe of architectural models, which we denote \mathcal{P} . Being abstractions of actual programs, an unbounded number of programs are effectively treated as equivalent. Thus, from this point in this article, we discard “programs” in their original sense and use the term with reference to architectural abstraction. Because we need not refer directly to programs in their original representation, we do not see a difficulty in using this alias.

Finally, Definition I allows us to use predicate calculus in our discussion of any formalism that can be mapped into this abstraction.

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 [29] is lack of *abstraction*, then there is more than enough room to improve the structural picture of software systems.

The elements of O-O architecture (listed in Section 3) support the view that, predominantly, design patterns characterize *static* attributes of this universe. Even many of the patterns that appear “dynamic” (“behavioural” patterns, as characterized in [26]) 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 structural, static relations. The examples provided in Section 4 illustrate this observation. Additional examples for this observation are provided in [21].

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,

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

$$\text{Factories} : \mathbf{P}(\mathbb{C}) \tag{4}$$

for instance, (3) states that the symbol `Draw` represents a specific method (“function member”), while (4) declares a variable that ranges over sets of classes.

As established by Perry and Wolf [1] (“the principle of least constraint”), 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 than detailing a concrete solution (program).

We conclude that we can carry an architectural specification as constraints on properties of participants and relations (Definition I) and formulated in logic formulae. This conclusion leads to the following definition:

Definition II: *Pattern*

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

Naturally, Definition II is too broad. It offers, however, a baseline for a formal discussion in specifications of patterns in different formal languages. In conjunction with Definition I, Definition II provides a basis for the formal definitions of many other useful terms, such as an *instance of a pattern* (Definition III) and *refinement of a pattern* (Definition VII).

2.3 Reasoning

A significant advantage to the use of a formal framework is in the ability to apply rigorous in-

ference rules so as to allow reasoning with the specifications and deriving 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 (intuitively speaking, the elements that partake in an implementation of π), and program that contains an instance of π .

It is important to distinguish between a pattern and an instance thereof. The following definition formulates this distinction:

Definition III: Instance of a Pattern

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

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

To illustrate Definition II, consider the following trivial “pattern”:

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

It is easy to show that the model in Example 1 contains an instance of (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:

$$\text{Invoke}(\text{BorderDecorator::Draw}, \text{Decorator::Draw}) \tag{6}$$

which is true in Example 1.

Corollary 1

From Definition I and Definition II 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*. Each rudiment is formalized in terms of the aforementioned definitions and illustrated using examples from [26]. The discussion is broken down along the distinction between *participants* and *collaborations* as the elements of patterns [26].

3.1 Participants

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

Rudiment A: Ground entities. The predominant agents in O-O software, as well as the *participants* of every single pattern in [26], 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:

$$\begin{aligned} &\text{Decorator, BorderDecorator, int} \in \mathbb{C} \tag{7} \\ &\text{Decorator}::\text{Draw}, \\ &\text{BorderDecorator}::\text{Draw} \in \mathbb{F} \end{aligned}$$

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).

Rudiment B: Uniform Sets. Observe that, 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. Here are a few examples: The set of *Visitor* classes in the *VISITOR*; the set of *operation-implementations* defined in the *BRIDGE*; the set of *concrete-strategy* classes that inherit from the abstract *strategy* class of the *STRATEGY* pattern; the set of *decorator* classes of the *DECORATOR* pattern; and so forth.

Similarly, higher order sets (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 IV: Uniform Set, Dimension

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

It is convenient to refer to a uniform set of dimension d as an entity 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*.

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 dispatching table. Formally:

Definition V: Clan

We say that a set of methods F is a *clan* in the set of classes 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 V in predicate calculus, which uses the relations *SameSignature* and *DefinedIn* can be found in [21]. 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 the same set of classes C . This abstraction is termed *tribe* and it can be found in almost every pattern [21].

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 (zero or more) occurrences of the relation \mathcal{R} .

Definition VI: Hierarchy

A 1-dimension class h is a *hierarchy* if the following condition holds:

$$\begin{aligned} \exists root \in h \bullet Abstract(root) \wedge \\ \forall cls \in h \bullet Inheritance^*(cls, root) \end{aligned} \quad (8)$$

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

Almost every pattern in [26] 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*.

Rudiment E: Ground Relations. We identify a small group of ground relations which, as illustrated in [30], 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 illustrates how ground relations model correlations and interactions among the elements of a program.

Abstract : $\mathbb{F} \cup \mathbb{C}$
Indicates whether the entity is abstract (in Eiffel: <i>deferred</i> ; methods in C++: <i>pure virtual</i>)
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:
<pre> new int number; (Java) int numbers[2]; (C++) if (string("A") == s) (C++) !INTEGER! num.make; (Eiffel) </pre>
DefinedIn : $\mathbb{F} \times \mathbb{C}$
Indicates that the method is a member in the class
Forward : $\mathbb{F} \times \mathbb{F}$
<i>Forward(f,g)</i> means that <i>f</i> invokes <i>g</i> with the additional requirement that the actual arguments in the invocation expression are the formal arguments of <i>f</i> . Such a method in C++ will look as follows:
<pre> void TCPConnection::ActiveOpen(int t) { _state->ActiveOpen(t); } </pre>
Invoke : $\mathbb{F} \times \mathbb{F}$
<i>Invoke(f,g)</i> indicates that <i>f</i> “may invoke” <i>g</i> , namely, that within the body of method <i>f</i> there is an expression whose evaluation invokes <i>g</i> . Note that we ignore the control flow
Inherit : $\mathbb{C} \times \mathbb{C}$
Indicates that the first-class inherits from the second
Reference [-To-Many] : $\mathbb{C} \times \mathbb{C}$
Indicates that left class defines an attribute whose type is of a single [multiple] instance(s) of the second class
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.

Table 1: Intuitive interpretations for ground relations

Rudiment F: Bijections. Consider the following quote from the specification of the *PROXY* [26, p. 270]: “each proxy operation ... forwards the request to the subject.” It implies that for every method *p* in the set *ProxyMethods* 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, as shown in Rudiment C, which we denote *Products*. Their collaboration is described as follows: “AbstractFactory ... defines a different operation for each kind of product it can produce.” [26, 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 [21].

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.

DisCo

Defined as an extension of *temporal logic of actions* [31], DisCo [32] consists of constructions that express the composition of classes and the semantics of methods. Although a temporal logic-

based formalism focuses on dynamic specifications, many specifications clearly map to our static framework. Consider for example its specification of the *OBSERVER* pattern [26]:

Extract 2: *OBSERVER* in DisCo [32]

```

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

```

The specification in Extract 2 describes the relationships between three classes and defines the semantics of one operation 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.

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

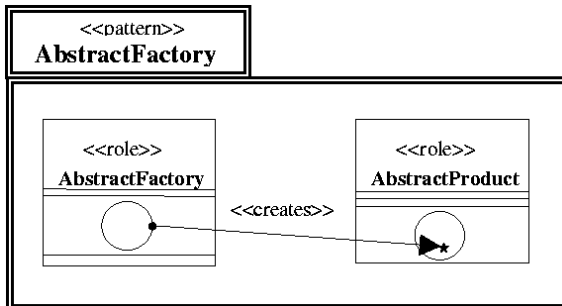
```

Reference(Subject, Data)
Reference(Observer, Data)
Assign(Attach, Subject, Observer)

```

Constraint Diagrams

Described as a precise visual specifications language combined with elements in UML [23], constraint diagrams [33] are a visual representation of first order set-theoretic relations. Extract 3 depicts the “role model” of the *ABSTRACT FACTORY* pattern.



Extract 3: “Role Diagram” of the *ABSTRACT FACTORY*

Again, 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.

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

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

LePUS

LePUS [21] is also defined as a visual language for the specification of O-O software architecture, yet its authors also define a symbolic equivalent for each well-defined diagram. Consider for example the symbolic specification of the *ABSTRACT FACTORY* in LePUS, which appears in Extract 4.

Extract 4: *ABSTRACT FACTORY* in LePUS

<pre> Factories : \mathbb{H} Products : \mathbf{P}(\mathbb{H}) FMs : \mathbf{P}(\mathbb{S}) </pre>
<pre> Create[↔](FMs ⊗ Factories, Products) Return Type[↔](FMs ⊗ Factories, Products) </pre>

Since LePUS is a highly concise language (*concision* is discussed in section 6), Extract 4 delivers more information than it may appear which results in a cumbersome formula. Hence, unlike previous extracts, the interpretation of Extract 4 is explained verbally by each one of its expressions:

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 $FMs \otimes Factories$ uses 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. LePUS defines the abbreviation $\mathcal{R}^{\leftrightarrow}(V, W)$ as an abbreviated form of a *bijection* (Rudiment F). Thus, the two statements below the dividing line indicate that for every factory method fm there is a product p such that
- $$\text{that is } \begin{array}{c} \text{Create}(fm, p) \\ \text{and} \\ \text{ReturnType}(fm, p). \end{array}$$

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.

Design Patterns' Formalisms

Formal Languages for the specification of design patterns are described in several publications. Helm, Holland and Gangopadhyay [34] 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 run time ("behavioural") characterizations.

Tool Support

Florijn, Meijers, and van Winsen [35] 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's leitmotif, and its arcs describe the roles of the connecting nodes. Predominantly, the *patterns' wizard* [36] 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 [37, 38, 39, 40, 41, 42, 43, 44], most of which are reviewed in [30].

Ground Relations

Other works have observed "elementary" relations between entities. Keller et. al [45] 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 rela-

tions between classes (e.g., *Inheritance*, *Reference*). Chiba [46] 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 ontology defined so far.

Refinement

Two articles [47; 48], describe one pattern as a "refinement" of another, meaning that one is a special case of the other. Cargil [49] describes "categories of patterns," where one category "refines" the other. Kim and Benner [50] describe the *push* and *pull models* as "refinements of the *OBSERVER* pattern" [26]. Similarly, Rohnert [51] describes "specializations of the *PROXY* pattern," such as *CACHE PROXY* and *PROTECTION PROXY*. The following definition formalises this intuition:

Definition VII: *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 \mathbb{M} , \mathcal{A} also satisfies π in \mathbb{M} .

By Definition VII, *refinement* is equivalent to *semantic entailment* [52]. 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 [48]. The article reports a debate between the authors of the *OBSERVER* pattern [26], which could not agree whether it is indeed refined by the *MULTICAST* (a pattern proposed by J. Vlisides). In [53], Eden, Gil, Hirshfeld and Yehudai show how the debate can be resolved using LePUS (Section 4).

Projection

The next relation we discuss extends the notion of *refinements* by means of manipulation of *dimension* (Definition IV). 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 VIII: *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 .

If this relation holds, we say that $\varphi(X)$ is an *abstraction* of x in $\varphi(x)$.

In [21] we illustrate how the *FACTORY METHOD* pattern results from a projection of the *Factory-Methods* and *Products* variables in the definition of the *ABSTRACT FACTORY* pattern. Example 2 summarizes this relation.

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:

<i>ABSTRACT FACTORY</i>	<i>FACTORY METHOD</i>	description
<i>Products</i> : $\mathbf{P}(\mathbb{H})$	<i>Products</i> : \mathbb{H}	A hierarchy vs. a set of hierarchies
<i>FMs</i> : $\mathbf{P}(\mathbb{S})$	<i>FMs</i> : \mathbb{S}	A signature vs. a set of signatures

6. Comparative Criteria

Having provided a common framework for architectural specifications in different languages, we now seek to compare the relative merits of each formalism that can be mapped to this framework. With the increase in the popularity of design patterns, we expect that even more formalisms should arise. The purpose of this section is to de-

fine properties that can be used as criteria in comparing between such formalisms.

We maintain that shorter expressions contribute to a clearer picture, of *concision* 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 IX: *Concision*

Let us assume a metric function that measures the length of expressions in \mathcal{L}_1 and \mathcal{L}_2 :

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

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 X: *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 IV)
- ◆ ground and set relations (Rudiment E and Rudiment F)
- ◆ clans (Definition V)
- ◆ hierarchies (Definition VI)

Observe that, while different formalisms may interpret ‘class’ and ‘method’ differently, Definition X 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:

Compactness

Definition X is satisfied by any language that assimilates the *rudiments* of section 3, such as higher order logic (i). 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 (ii). 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 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. In other words, 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” leitmotifs by providing a measurement for *compactness* at this point.

Dynamic specifications. Despite the achievement in specification through static properties, our frame of reference does not allow the expression of certain behavioural conditions. Formalisms that may apply are Temporal Logic of Actions [54], as in [32], and Abstract State Machines [55].

Refining and completing the criteria. Additional constraints can be suggested to refine and improve the criteria we suggested. Compactness, for instance, may be defined as a relation between languages.

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 comparing 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.

Acknowledgements

Special thanks to Dr. Björn Victor of the Department of Computer Systems, Uppsala University, for his insights and detailed comments. Many thanks also to Peter Grogono for his feedback. We thank Marie Gerrard for her support.

i A construction of these elements in higher order logic appears in [30].

ii UML is an example that comes to mind.

Biographies

Dr. Amnon H. Eden is with the department of computer science in Concordia University, Montreal, Canada. Dr. Eden developed the LePUS specification language for object oriented design and architecture. His research interest include, but are not limited to: Object-oriented design and programming, software architecture, formal and visual specification of software, and design patterns. Since 1992, Dr. Eden designed, programmed, and lead the development of software. During the years 1997-2000 Dr. Eden has chaired a software engineering transition programme and was a guest lecturer in various departments. He can be reached at eden@acm.org.

Dr. Yoram Hirshfeld is with the department of pure mathematics in Tel Aviv University. Dr. Hirshfeld has publications in the areas of mathematical logic and non-standard analysis. His research interests include computability, set theory, and nonstandard models. He can be reached at joram@post.tau.ac.il.

References

- [1] D. E. Perry, A. L. Wolf (1992). "Foundation for the Study of Software Architecture". *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52.
- [2] N. Medvidovic, R. N. Taylor (1997). "A Framework for Classifying and Comparing Architecture Description Languages". In *Proceedings of the Sixth European Software Engineering Conference*, together with *Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 22-25, 1997.
- [3] C. A. R. Hoare (1985). *Communicating Sequential Processes*. New Jersey: Prentice-Hall.
- [4] R. Allen, D. Garlan (1997). "A Formal Basis for Architectural Connection." *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, 1997, pp. 213-249.
- [5] T. R., Dean, J. R. Cordy (1995). "A Syntactic Theory of Software Architecture". *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.
- [6] G. Abowd, R. Allen, D. Garlan (1993). "Using Style to Understand Descriptions of Software Architecture". *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1993.
- [7] J. M. Spivey (1989). *The Z Notation: A Reference Manual*. New Jersey: Prentice-Hall.
- [8] D. Harel (1987). "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, Vol. 8, No. 3, June 1987, pp. 231-274.
- [9] C. A. Petri (1962). "Communications with Automata". Technical report RADC-TR-65-377, Applied Data Research, Princeton, N. J.
- [10] I. Craig (1999). *The Interpretation of Object-Oriented Programming Languages*. Berlin: Springer-Verlag.
- [11] G. Odenthal, K. Quibeldey-Cirkel (1997). "Using Patterns for Design and Documentation". *Proceedings of the European Conference of Object Oriented Programming 1997*. Lecture Notes in Computer Science. Berlin: Springer.
- [12] J. Coplien, D. Schmidt, eds. (1995). *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.
- [13] J. M. Vlissides, J. O. Coplien, N. L. Kerth (1996). *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley.
- [14] R. Martin, D. Riehle, F. Buschmann, eds. (1997). *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley.
- [15] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann (2000). *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, Ltd.
- [16] pattern-discussion. Mailing list: <http://hillside.net/patterns/List s.html>
- [17] gang-of-four-patterns. Mailing list: <http://hillside.net/patterns/List s.html>
- [18] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fixdahl-King, S. Angel (1977). *A Pattern Language*. Oxford University Press, New York.
- [19] C. Alexander (1979). *The Timeless Way of Building*. Oxford University Press, New York.
- [20] A. H. Eden. "Giving 'The Quality' a Name." *Journal of Object Oriented Programming*, Vol. 11, No. 3, June 1998, pp. 5-11.
- [21] A. H. Eden (2000). "Precise Specification of Design Patterns and Tool Support in Their Application," PhD diss., Department of Computer Science, Tel Aviv University.
- [22] K. Popper (1969). *Conjectures and Refutations*. London: Rutledge.
- [23] G. Booch, I. Jacobson, J. Rumbaugh (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

-
- [24] G. Booch (1994). *Object Oriented Analysis and Design With Applications*. 2nd edition. Benjamin/Cummings.
- [25] D. G. Firesmith, I. Graham, B. Henderson-Sellers (1998). *Open Modeling Language (Olm) Reference Manual*. Cambridge University Press.
- [26] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA: Addison-Wesley.
- [27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenzen (1991). *Object Oriented Modeling and Design*. Prentice Hall.
- [28] J. Gosling, B. Joy, G. Bracha (2000). *The Java™ Language Specification*, 2nd edition. Reading, MA: Addison Wesley Longman, Inc.
- [29] W. W. Gibbs. "Software's Chronic Crisis." *Scientific American*, September 1994, pp. 86-95.
- [30] A. H. Eden, Y. Hirshfeld, A. Yehudai (1999). "Towards a Mathematical Foundation for Design Patterns". Technical report 1999-004, Department of Information Technology, Uppsala University.
- [31] L. Lamport (1994). "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 872-923.
- [32] T. Mikkonen (1998). "Formalizing Design Patterns." *Proceedings of the International Conference on Software Engineering*, April 19 - 25, 1998, pp. 115-124. Kyoto, Japan.
- [33] A. Lauder, S. Kent. "Precise Visual Specification of Design Patterns." In *Proceedings of the 12th European Conference on Object Oriented Programming, Brussels, Belgium*, July 1998. Lecture Notes in Computer Science 1445. Jul, Eric (ed.) Berlin: Springer-Verlag.
- [34] R. Helm, I. M. Holland, D. Gangopadhyay (1990). "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 21 - 25, 1990, Ottawa, Canada, pp. 169-180.
- [35] G. Florijn, M. Meijers, P. van Winsen (1997). "Tool Support in Design Patterns". In: Askit M., S. Matsuoka (1997, eds.) *Proceedings of the 11th European conference on Object Oriented Programming - ECOOP 97*. Lecture Notes in Computer Science no. 1241. Berlin: Springer-Verlag.
- [36] A. H. Eden, J. Gil, A. Yehudai (1997). "Precise Specification and Automatic Application of Design Patterns." *Proceedings of the Twelve IEEE International Automated Software Engineering Conference (ASE 1997), Lake Tahoe, Nevada*, Nov. 3-5, 1997, pp. 143-152. Los Alamos: IEEE Computer Society Press.
- [37] P. P. Pal. "Law-Governed Support for Realizing Design Patterns." *Proceedings of 17th Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17)*, 1995. New Jersey: Prentice Hall.
- [38] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, P. S. Yu. "Automatic Code Generation from Design Patterns." *IBM Systems Journal*, Vol. 35, No. 2, 1996, pp. 151-171.
- [39] Quintessoft Engineering, Inc. (1997). C++ Code Navigator 1.1. <http://www.quintessoft.com>
- [40] K. Brown (1996). "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk." M. Sc. thesis, University of Illinois.
- [41] P. Alencar, D. Cowan, J. Dong, C. Lucena. "A Pattern-Based Approach to Structural Design Composition." *IEEE 23rd Annual International Computer Software and Application Conference*, October 1999, pp. 160-165.
- [42] C. Kramer, L. Prechelt. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software." *Proceedings of the Working Conference on Reverse Engineering*, November 8-10, 1996, Monterey, CA, pp. 208-215.
- [43] J. Bosch. "Relations as Object Model Components." *Journal of Programming Languages*, Vol. 4, No. 1, 1996, pp. 39-61.
- [44] M. O'Cinnéide, P. Nixon. "A Methodology for the Automated Introduction of Design Patterns." *Proceedings of the IEEE International Conference on Software Maintenance*, 30 August - 3 September, 1999.
- [45] R. K. Keller, R. Schauer, S. Robitaille, P. Page. "Pattern-based Reverse Engineering of Design Components." In: *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 226-235, Los Angeles, CA, May 1999. IEEE.
- [46] S. Chiba (1995). "A Metaobject Protocol for C++." *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 15 - 19, 1995, pp. 285-299. Austin, USA.
- [47] E. Agerbo, A. Cornils. "How to Preserve the Benefits of Design Patterns." *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 18 - 22, 1998, Vancouver, Canada, pp. 134-143.
- [48] J. Vlissides (1997). "Multicast". *C++ Report*, Sep. 97. SIGS Publications.
- [49] T. Cargill (1996). "Localized Ownership: Managing Dynamic Objects in C++". In: J. M. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.

-
- [50] J. J. Kim, K. M. Benner (1996). "Implementation Patterns for the Observer Pattern". In: J. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.
- [51] H. Rohnert (1996). "The Proxy Design Pattern Revisited". In: J. Vlissides, J. O. Coplien, N. L. Kerth (1996, eds.) *Pattern Languages in Program Design 2*. Addison-Wesley.
- [52] M. R. Huth, M. D. Ryan (2000). *Logic in Computer Science: Modeling and Reasoning About Systems*. Cambridge, UK: Cambridge University Press.
- [53] A. H. Eden, Y. Hirshfeld, A. Yehudai. "Multicast - Observer \neq Typed Message." *C++ Report*, Vol. 10, No. 9, October 1998, pp. 33-39.
- [54] L. Lamport (1994). "The Temporal Logic of Actions." *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 872-923.
- [55] Y. Gurevich (1994). "Evolving Algebras." In: B. Pehrson, I. Simon (eds.). *IFIP 13th World Computer Congress 1994*, Volume I: Technology and Foundations, 423-427.