On the Definitions of Architecture, Design, and Implementation ([*])

Amnon H. Eden
Center for Inquiry, Amherst, NY, and
Department of Computer Science,
University of Essex, UK
`eden@acm.org`

Rick Kazman
Software Engineering Institute, Pittsburgh, PA
and *University of Hawaii, Honolulu, HI*
`kazman@sei.cmu.edu`

Abstract

The terms architecture, design, *and* implementation *are typically used informally in partitioning software specifications into three coarse strata of abstraction. But these strata are not well-defined in either research or practice and often overlap causing confusion and needless discussion.*

To remedy this problem we formally define two criteria: the Intension *and the* Locality *Criteria, and show that the intuitive discrimination between the three terms architecture, design, and implementation is qualitative and not merely quantitative. We demonstrate that architectural styles are* intensional *and* non-local*; that design patterns are* intensional *and* local*; and that implementations are* extensional *and* local*.*

1. Introduction

In their seminal article, Perry and Wolf [31] developed "an intuition about software architecture through analogies to existing disciplines." Building on this, Shaw and Garlan [39] suggest that "software architecture involves the description of elements from which systems are built." A considerable body of work, stemming back to DeRemer and Kron's *module interconnection languages* (MIL) [8], focuses on the specification, construction, and analysis of large software systems defined by these terms (e.g., [34], [27], [16]). Architecture description languages (ADL) combine a formal specification language with tools supporting the construction and analysis of software systems from such specifications.

Seeking to separate *architectural design* from other design activities, definitions of *software architecture* stress the following:

- "*architecture* is concerned with the selection of architectural elements, their interaction, and the constraints on those elements and their interactions… *Design* is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements." [31]
- Software Architecture is "concerned with issues ... beyond the algorithms and data structures of the computation." [17]
- "architecture … is specifically not about … details of implementations (e.g., algorithms and data structures.) … Architectural design involves a richer collection of abstractions than is typically provided by OOD." [30]
- "Architecture ? Design? … Design is an activity. Architecture, or architectural design, is design at a higher level of abstraction." [23]
- Architecture focuses on the externally visible properties of software "components." [2]

In suggesting typical "architectures" and "architectural styles", existing definitions consist of examples and offer anecdotes rather then provide unambiguous, clear notions.

In practice, the terms "architecture", "design" and "implementation" appear to connote milestones in a continuum between complete details ("implementation"), few details ("design"), and the highest form of abstraction ("architecture"). But the amount of detail alone is insufficient to characterize the differences, because architecture and design documents often contain information that is not explicit in the implementation (e.g., design constraints, standards, performance goals) and therefore they cannot result from mere omission of detail. A clear distinction has remained elusive and this lack of distinction is the cause of much muddy thinking, imprecise communication, and wasted, overlapping effort.

Confusion inevitably arises from this imprecision, and *architecture* is often used as a mere synonym for *design*. For example, the "Siemens" catalogue [4] defines "architectural patterns" that are in par with "design patterns" defined by the "Gang of Four" [15].

Confusion also stems from the use of the same specification language for both architectural and design specifications. For example, the *Software Engineering Institute* (SEI) classifies UML [3] as an architectural description language [38], and it has become the industry de facto standard ADL, although UML was specifically designed

[*] Technical report, January 2003, Dept. of Computer Science, University of Essex, United Kingdom.

to manifest detailed design decisions (and this is its most common use).

Confusion also exists with respect to the artifacts of design and implementation. UML *class diagrams* [3], for instance, are a prototypical artifact of the design phase. Nonetheless, class diagrams may accumulate enough detail to allow code generation of very detailed programs, an approach that is promoted by CASE tools such as *Rational Rose*® [36] and *System Architect*® [33]. Using the same specification language further blurs the distinction between artifacts of the design (class diagrams) from the implementation (source code.)

Why are we interested in such distinctions? With time, terms that are used interchangeably lose their meaning and end up as mere platitudes, resulting inevitably in ambiguous descriptions given by developers, and significant effort is wasted in discussions of the form "by design I mean… and by architecture I mean…"

The contribution of this paper is to provide insight on the largely-informal dialectic by appealing to both intuition and to formal ontology. By putting these terms on a solid footing not only do we disambiguate the progressively murky discourse in "architectural specifications" but provide a foundation for formal reasoning and analysis, as well as a firm foundation for informal "chalk-talk" discussions. Finally, tools supporting design and architectural specifications, where intuitive perceptions are insufficient, will benefit by accurately defining this distinction.

1.1 The Intension/Locality Thesis

The term "abstraction" has been used in many different contexts with various interpretations. To elucidate the relationship between *architecture*, *design*, and *implementation*, we distinguish at least two separate interpretations for *abstraction* in our context:

1. *Intensional* (vs. *extensional*) specifications are "abstract" in the sense that they can be formally characterized by the use of logic variables that range over an unbounded domain;
2. *Non-local* (vs. *local*) specifications are "abstract" in the sense that they pervade *all* parts of the system (as opposed to being limited to some part thereof).

Both of these interpretations turn out to be critical in distinguishing among the terms *architecture*, *design*, and *implementation*, which we jointly refer to as the *intension/locality thesis*:

(i) Architectural specifications are *intensional* and *non-local*;
(ii) Design specifications are *intensional* but *local*; and
(iii) Implementation specifications are both *extensional* and *local*.

The intension/locality thesis is summarized, for easier reference, in Table 1.

Table 1. The Intension/Locality Thesis

Architecture	*Intensional*	*Non-local*
Design	*Intensional*	*Local*
Implementation	*Extensional*	*Local*

1.2 Clarification and Structure of This Paper

The intension/locality thesis can be understood correctly only in the context of the ontology defined in the following section. In Section 0 we define *design models*, which are crucial to the remainder of out discussion. Design models are abstractions which provide the underlying "meaning" of programs. Design models are associated with programs using a "meaning" (*denotation*) function, which allow us to determine whether a specification is "satisfied" by a program.

In Section 3, we formally define the Intension criterion and the Locality criterion. We distinguish our interpretation for "intensionality" from the accepted usage, as we define it in terms of the "meaning" (denotation) that programs have, i.e., in terms of *design models*.

Sections 4, 5, and 6 provide case studies in applying the Intension and Locality criteria using our formal ontology. In Section 4 we demonstrate that implementations any programming language, including generics and C++ templates are *extensional* and *local*. In Section 5 we show that design patterns such as the Factory Method and design specifications such as the *Enterprize JavaBeans*™ and Java™ *Swing*'s MVC are *intensional* and *local*. In Section 6 we demonstrate that architectural styles such as Pipes and Filters and Layered Architecture are *intensional* and *non-local*, , and so is the Law of Demeter.

In Section 7, we discuss some of the ramifications of our criteria. The discussion in UML class diagrams reveals that, indeed, class diagrams have a separate place in the hierarchy of abstractions we describe. Section 8 summarizes the contributions of this paper

2. Setting the Scene

In this section, we provide the underlying formal ontology that underlies the *Intension*/*Locality* criteria.

2.1 Design Models

Turing [43] and *random-access machines* [5] provide robust computational models suitable for reasoning about algorithms. Other computational models and formalisms (e.g., Petri nets [32], statecharts [20], and temporal logic [24]) facilitate reasoning about certain behavioral specifications.

The discussion in architectural and design specifications, however, involves reasoning on constructs such as *classes*, *methods*, and *function calls*. Most other formalisms incorporate too many implementation detail and do not allow a discussion in the appropriate level of abstraction. As we seek to establish the relation between architectural or design specifications and implementations, we base our discussion on a formalism that abstracts programs to a more convenient representation.

On par with *evolving algebras* [19], Eden and Hirshfeld [12] demonstrate how to model source code as *design models*, which are first order, finite *structures* in mathematical logic [1]:

Definition I*. Let* m *designate the pair* $\langle \mathbb{U}_{\mathsf{m}}, \mathbb{R}_{\mathsf{m}} \rangle$*, such that* $\mathbb{U}_{\mathsf{m}} = \{a_1, \ldots a_k\}$ *is a finite set of atoms, and* $\mathbb{R}_{\mathsf{m}} = \{\mathcal{R}_1, \ldots \mathcal{R}_n\}$ *is a finite set of ground relations amongst these atoms.*

We say that m *is a* ***design model****. The set of all design models is designated* $\mathcal{M}$.

Table 2. A Java™ program and its denotation (from [11].)

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

The design model of this program consists of the following:

Atoms:

$\mathbb{C} = \{$`Decorator`, `BorderDecorator`, `int`, `void`$\}$

$\mathbb{F} = \{$`BorderDecorator.Draw`, `Decorator.Draw`$\}$

Relations:

Abstract(`Decorator`*)*

Member(`Decorator.Draw`, `Decorator`*)*

Member(`BorderDecorator.Draw`, `BorderDecorator`*)*

Inherit(`BorderDecorator`, `Decorator`*)*

Reference(`BorderDecorator`, `int`*)*

Invoke(`BorderDecorator.Draw`, `Decorator.Draw`*)*

ReturnType(`Decorator.Draw`, `void`*)*

ReturnType(`BorderDecorator.Draw`, `void`*)*

Table 2 depicts a detailed example of a trivial Java™ program and a design model that represents it. As this example demonstrates, an object-oriented program is abstracted as a collection of definitions of *classes* and *methods* (also *routines* or *function members*) and their relations. *Atoms* represent *classes* and *methods* declared in the program, such as the class `Decorator` and the method `Decorator.Draw`. *Relations* represent their correlations, such as

Inherit(`BorderDecorator`,`Decorator`*)*

Invoke(`BorderDecorator.Draw`,`Decorator.Draw`*)*

Note that *design models* are abstractions which were made to reflect only the structural aspects of computer programs that are relevant to design and architecture. Hence, our analysis focuses on the declarations of program constructions. Obviously, this representation limits the type of reasoning we may perform (e.g., for discussing *fairness* [24]), but it is appropriate for the purposes of our discussion.

2.2 Specifications, Denotations, and Programs

In this subsection, we discuss specifications, programs, and their relations. We make some reasonable assumptions on the languages used to write specifications. These assumptions allow us to provide a clear definition for the *intensional* criterion (Definition VI).

Let us designate $\mathcal{SPEC}$ as the set of formal languages of any order [1]. Let $\mathcal{SPEC}$* designate the set of all expressions made in some language in $\mathcal{SPEC}$. A **specification** is an element of $\mathcal{SPEC}$*.

$\mathcal{SPEC}$ includes familiar specification languages such as $\mathbb{Z}$ [40], as demonstrated in formulas (3.1) and (3.2), and LePUS [9], as demonstrated in formula (2). $\mathcal{SPEC}$ also includes programming languages such as Eiffel [28], C++ [41], and Java™ [18]. Naturally $\mathcal{SPEC}$ is not restricted to known programming or specification languages.

A specification is only useful if we can determine whether it is "satisfied" or not. Having chosen *design models* as our semantics we can ask: Does this model "satisfy" our specification? More importantly, we would like to be able to answer the question: Does this program satisfy our specification?

To answer these questions, we first define an "instance" of a specification:

Definition II*. Let* $\varphi(x_1, \ldots x_n)$ *be a first order expression in* $\mathcal{SPEC}$**, such that* $x_1, \ldots x_n$ *are free variables in* φ*. Let* m *designate a design model (Definition I) containing the n-tuple of atoms* $(a_1, \ldots a_n)$*. Let* $\mathcal{A}$ *be the consistent assignment* [1] *of* $a_1, \ldots a_n$ *to* $x_1, \ldots x_n$*.*

If the result of assignment $\mathcal{A}$ in φ is true in m then we say that $(a_1,\ldots a_n)$ is "an ***instance*** *of φ in the context of $\mathcal{A}$". If there exists such an assignment $\mathcal{A}$, we say that "m* ***instantiates*** *φ", written $\mathsf{m}\models\varphi$.*

Definition II extends naturally to n-tuples of sets of atoms of any order, and to include expressions in higher order languages, such as LePUS.

Observe that, in the degenerate case where φ has no free variables (also *closed formula, sentence*), φ is either true or false in each model m. This distinction will serve us in defining the Locality criterion (Definition VIII).

What is the expected relation between a program and an instance? An instance is only a part of the program, and depending on the specification, every program can contain zero, one, or any number of instances. In addition, we expect a "program" to be as a specification that is associated with only one *design model*. The association between "real" programs and design models is provided by a morphism we refer to as the *denotation function*:

Definition III. *Let D : $\mathcal{SPEC}^* \rightarrow \mathcal{M}$ designate a relation that maps each element in a subset of $\mathcal{SPEC}^*$ into a design model, such that for every expression φ in $dom(\mathsf{D})$ the domain of D, the following conditions hold:*

- *There is exactly one design model m_φ such that*

  $\mathsf{m}_\varphi\models\varphi$ *(m_φ instantiates φ)*
- $\mathsf{D}(\varphi)=\mathsf{m}_\varphi$ *(D maps φ to m_φ)*

We say that D is a ***denotation function***.

The Java™ example in Table 2 and the C++ example in Table 3 demonstrate typical denotations.

Observe that we expect $dom(\mathsf{D})$ to include only a *small subset* of the expressions in $\mathcal{SPEC}^*$. Obviously, there are expressions in $\mathcal{SPEC}^*$ that are true in more than one design model, as well as expressions are not true in any design model.

Definition IV. *Based on Definition III, we introduce the following nomenclature:*

$\mathcal{P}_{\mathsf{D}}$	The set of all *programs* (the domain of D)
$[\![\pi]\!]_{\mathsf{D}}$	$\mathsf{D}(\pi)$
program	An element of $\mathcal{P}_{\mathsf{D}}$

Note that only *programs* have denotations (one denotation for every program).

By Definition IV, there is only one possible denotation to each *program* π. Thereof, we may refer to $[\![\pi]\!]_{\mathsf{D}}$ as "*the* design model (the denotation) of π according to D." The converse, however, is not true, and any number of programs in $\mathcal{P}_{\mathsf{D}}$ can be denoted by (i.e., mapped by D to) a single design model (also D is not *one-to-one* function.) This conforms to our view of D as means of abstracting programs. Additional examples for *denotations* for O-O programming languages are provided in [9].

In the reminder of our discussion, unless specified otherwise, we assume a fixed *denotation* D, defined along the lines as demonstrated in Table 2. Thus, we are free to speak of "a program π" and of "the design model (denotation) of π", marked $[\![\pi]\!]$.

We now have a well-defined notion of programs and specifications. In combination with the definition of an "instance" of a specification, we can conclusively determine whether a program satisfies a given specification:

Definition V. *Let φ designate a specification. Let π designate a program. We say that π* ***satisfies*** *φ iff $[\![\pi]\!]\models\varphi$, namely, iff the design model of π instantiates φ.*

In the following sections we will set apart: architecture, design, and implementation specifications based on observing properties of the groups of programs that satisfy each specification

3. The *Intension*/*Locality* Criteria

We will now define the concepts of *intension* and *locality*. In the following sections, we will apply these criteria, both formally and informally, to distinguish between *architectural specifications*, *design specifications*, and *implementations*.

3.1 The Intension Criterion

Perry and Wolf [31] have established that architectural specifications must be made in intensional terms. Speaking of the desired properties of an ideal specification language for software architecture they write: "We want a means of supporting a 'principle of least constraint' to be able to express only those constraints in the architecture that are necessary at the architectural level of the system description". It constrains only what it needs to, in terms of properties imposed over free variables.

Traditionally, *intensional specifications* define a concept via a list of constraints. For example, mathematical concepts are usually defined intensionally. For instance: "A *prime number* is a number that divides only by itself and by the number 1". In contrast, NATO is an organization that is defined extensionally, namely, by itemizing its members: United States, United Kingdom, France, and so forth.

Using the distinction made by Immanuel Kant [22], we treat program as an *analytic* notion, not *synthetic,* that is, similar to a mathematical concept. In these terms, we say that this paper constitutes of analytical reasoning (the Intension/Locality criteria) to empirical manifestations of selection design patterns and architectural styles.

The notion of intensionality that we define here diverges slightly from the philosophical concept. We say that a specification is intensional if and only if it has an unbounded number of *instances* (Definition II):

Definition VI*. We say that a specification is* ***intensional*** *iff there are infinitely-many possible instances (Definition II) thereof. Conversely, all other expressions are* ***extensional****.*

The corollary that follows establishes the intuition that, given infinitely many instances, there should also be infinitely many *design models* that "satisfy" the specification.

Corollary 1*. An intensional specification can be instantiated (Definition II) by a non-finite number of design models.*

Proof*: According to Definition I and Definition III, every element in the range of the denotation function is a finite structure. Thus, for any given formula φ and design model m, m may contain at most a finite number of instances to φ. Thus, to allow an infinite number of instances to φ, there must be a non-finite number of design models that instantiate φ.*

?

Following the 'principle of least constraint', an architectural specification, must have an unbounded number of instances, or using our terms, is expected to be "intensional". The same applies to design patterns, as demonstrated in sections 5.2 and 5, respectively. But what about other forms of specifications?

Prima facie, it appears that some programming specifications (such as C++ templates and Eiffel generics) might also be intensional. This is not true in the context of *design models*: As we show in section 4 with detail, specifications in any programming language, including generics and interpreted code are, under the assumptions provided in Definition I and Definition III, extensional. This is an important point: implementations are extensional and this alone distinguishes them from design and architecture specifications. The remaining distinction, between design and architecture, is one of locality, which is explored next.

3.2 The Locality Criterion

Monroe et. al [30] argue that "Architectural designs are typically concerned with the entire system." Similarly, we observe that an architectural style that pervades a system [17] manifests a property that is shared across modules of the system. This intuition motivates the *locality* criterion: It captures the intuition that a design specification that is restricted only to part(s) of the system does not reflect an architectural property. Consequently, if there are modules that do not satisfy a certain constraint, then either the constraint is not architectural in this program, or else these modules are not part of the same "program".

As a simple example, consider applications designed with a "universal base class". Although the language does not require it, several C++ class libraries (e.g., NIHCL and Microsoft's MFC) are constructed by this rule. Formally, this property can be expressed as follows:

$$\forall c \bullet Class(c) \Rightarrow Inherit^*(c, \texttt{Object}) \qquad (1)$$

(where $Inherit^*$ is the transitive closure of the binary relation $Inherit$.) The intension/locality thesis argues that formula (1) is *architectural* not only because it is intensional but also because it pervades *all parts* of the system. In our example, *any* class must be bound to `Object`, so this clearly has architectural implications.

Subsumption. The locality criterion requires the notion of *subsumption* relation between two structures, which requires the following definition:

Definition VII*. Let* $\mathsf{m} = \langle \mathbb{U}_\mathsf{m}, \mathbb{R}_\mathsf{m} \rangle$, $\mathsf{n} = \langle \mathbb{U}_\mathsf{n}, \mathbb{R}_\mathsf{n} \rangle$ *be design models (Definition I). We say that* n ***subsumes*** m*, written* $\mathsf{m} \preceq \mathsf{n}$*, iff the following conditions hold:*

- $\mathbb{U}_\mathsf{m} \subseteq \mathbb{U}_\mathsf{n}$
- *For every relation* $\mathcal{R} \in \mathbb{R}_\mathsf{m}$ *there exists a relation* $\bar{\mathcal{R}} \in \mathbb{R}_\mathsf{n}$ *such that* $\mathcal{R} \subseteq \bar{\mathcal{R}}$*.*

Informally, we say that model n *subsumes* model m if m is a "submodel" of n. We can also view n as an "extension" to m, not unlike "strict inheritance" [42].

Definition VIII*. We say that a specification* φ *is* ***local*** *iff the following condition holds:*

If φ *is satisfied in some design model* m *then it is satisfied by every design model that subsumes* m*.*

We will now use the *Intension* and *Locality* criteria to illustrate the difference between programs, design specifications, and architecture specifications.

4. Programs

The definition of *instance* and the precise expression to the *Intension* criterion (Definition VI) allows us to prove part (iii) of the intension/locality thesis:

The lemma of "extensions"*: Programs are extensional specifications.*

Proof: *Let* π *be an element of* $\mathcal{P}_\mathsf{D}$ *. Let us assume by negation that* π *is intensional. By Corollary 1,* π *can be* *instantiated by a non-finite number of distinct design models. Let us designate this set as* P. *By Definition **II**,* π *is true in every model in* P. *However, by Definition III, there is only one design model such* π *is true therein,* $[\![\pi]\!]_{\mathsf{D}}$. *This contradicts with* P *in our assumption. Therefore,* π *cannot be intensional.*

?

One may wonder how the *lemma of extensions*, phrased to match the wording of the intension/locality thesis, is different from the very definition of a *program* (Definition IV). The subtlety that needs consideration is the difference between one *instance* of a formula vs. a design model that incorporates such an instance (i.e., *satisfies* the formula.)

***Corollary 2**. C++ templates are extensional.*

We illustrate the proof to this corollary using the design model of a C++ program with templates, shown in Table 3. The general proof follows directly from this example.

Table 3. A C++ program and its denotation

```
template <class C> class Stack
   {/* ... /*}

int main() {
   Stack<int> si;
   return 0;
}
```

This program is interpreted by only one design model, which consists of the following:

Atoms:

$\mathbb{C} = \{\texttt{Stack, si}\}$

$\mathbb{F} = \{\texttt{main}\}$

Relations:

Generic(`Stack`*)*

Instantiate(`Stack, si, int`*)*

Return(`main, int`*)*

This example illustrates that, while C++ templates may be viewed as intensional with respect to other semantic frameworks, the ontology we have provided assigns each program with just one “interpretation”. The reason is that the formal semantics of a program (its *interpretation*) is defined by the respective *design model* and not otherwise, e.g., by the machine code generated from its compilation. Thus, a template is a concrete construct even if it can be used to define other concrete constructs, since the notions of *design models* and of a *denotation* we provided guarantee that expressions in every conventional programming language are extensional.

5. Design Specifications

Design specifications in industry are commonly described informally. For the purpose of our discussion, we use formal specification of widely used design patterns and class libraries.

5.1 Factory Method

The following example is drawn from the published patterns literature. This allows us to test our ideas on some of the most widely published and used design specifications.

According to the intension/locality thesis, we expect design pattern specifications to be local and intensional. First we will develop an intuition about this claim and then formalize it.

Coplien and Schmidt [6] argue that “design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context”. Stripped from the context of a particular application, design patterns represent *categories* of solutions, each pattern has an unbounded number of implementations (as implied by the very choice of the name “pattern”). Thus they are expected to be *intensional*.

Design patterns are commonly perceived as “less abstract” than architectural specifications. For example, they are commonly referred to as “microarchitectures” [37], that is, as if they were like architectures that only apply to a limited module. Using our terminology, we thus expect them to be *local*.

Consider, for example, the Factory Method design pattern [15]. Essentially, the pattern’s solution offers three sets of participants:

1. A set of *product classes*
2. A set of *factory classes*
3. A set of *factory methods*

The collaborations between these participants are constrained as follows:

4. All *factory methods* share the same signature (thereby allowing for dynamic binding), and each is defined in a different *factory* class.
5. Each *factory method* produces instances of exactly one *products* class.

Figure 1 illustrates the general notion of the pattern. Observe that the set of $\langle$*factory-i, factory-method-i, product-i*$\rangle$ triplets is unbounded, because the number of possible *factory* and *product* classes is not bounded.

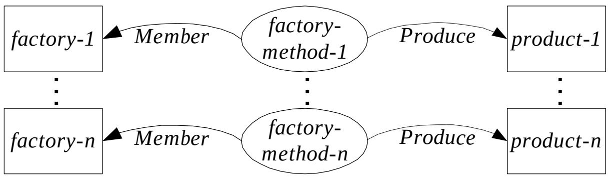


Figure 1. The general structure of the "Factory Method" pattern

For our discussion in design patterns we use LePUS, a formal specification language for object-oriented design. A detailed definition of the language appears in [9]. The five parts of the Factory Method definition are formally expressed by expressions (2.1) to (2.5) as follows:

$$Products : \mathbf{P}(\mathbb{C}) \quad (2.1)$$

$$Factories : \mathbf{P}(\mathbb{C}) \quad (2.2)$$

$$FactoryMethods : \mathbf{P}(\mathbb{F}) \quad (2.3)$$

$$Clan(FactoryMethods,Factories) \quad (2.4)$$

$$Produce^{\leftrightarrow}(FactoryMethods,Products) \quad (2.5)$$

Expressions (2.1) through (2.3) declare two sets of *classes* and one set of *methods*. Expression (2.4) indicates that the pair $\langle FactoryMethods,Factories \rangle$ satisfies the predicate $Clan$, meaning that –

1. All "factory methods" share the same signature (i.e., they share the same dispatch table);
2. The relation $Member$ is a bijection (i.e., *one-to-one* and *onto* function) between the sets $FactoryMethods$ and $Factories$.

Finally, expression (2.5) indicates that the ground relation $Produce$ is a bijective function between the set $FactoryMethods$ and the set $Products$.

It is evident that the composite expression (2.1) to (2.5), designated ϕ?below, is *intensional.*

Corollary 3*. "Factory method" is local.*

Proof: *We will show that, given any design model* m *such that* $\overset{\ni}{\phi}$ *is true therein, then* $\overset{\ni}{\phi}$ *is true in any design model that subsumes* m.

Let $\mathsf{m} = \langle \mathbb{U},\mathbb{R} \rangle$ *be a design model such that* $\overset{\ni}{\phi}$ *is true therein. The complete specification of* $\overset{\ni}{\phi}$ *is thus:*

$$\begin{aligned}
&\exists Products,Factories,FactoryMethods \bullet\\
&\quad (\forall m1,m2 \in FactoryMethods \bullet\\
&\quad\quad SameSignature(m1,m2)) \wedge\\
&\quad (\forall m \in FactoryMethods\ \exists f \in Factories \bullet\\
&\quad\quad Member(m,f)) \wedge\\
&\quad (\forall f \in Factories\ \exists m \in FactoryMethods \bullet\\
&\quad\quad Member(m,f)) \wedge\\
&\quad (\forall m \in FactoryMethods\ \exists p \in Products \bullet\\
&\quad\quad Produce(m,p)) \wedge\\
&\quad (\forall p \in Products\ \exists m \in FactoryMethods \bullet\\
&\quad\quad Produce(m,p))
\end{aligned}$$

Since $\overset{\ni}{\phi}$ *is true in* m, $\mathbb{U}_{\mathsf{m}}$ *must contain three sets of atoms,* $p_1,\ldots p_n$, $f_1,\ldots f_n$, *and* $m_1,\ldots m_n$, *such that they satisfy the existentially-quantified variables* $Products$, $Factories$, *and* $FactoryMethods$ *in* $\overset{\ni}{\phi}$, *respectively. Thus, we infer the following:*

1. $SameSignature(m_i,m_j)$ *holds for every pair of methods* m_i,m_j*;*
2. *There exists n "triplets" in the form* $\langle p_i,f_j,m_k \rangle$ *such that:*
   - $Member(m_k,f_j)$
   - $Produce(m_k,p_i)$
3. *Every element of one of the sets* $p_1,\ldots p_n$, $f_1,\ldots f_n$, *and* $m_1,\ldots m_n$ *occurs in exactly one "triplet".*

Let $\mathsf{n} = \langle \mathbb{U}_{\mathsf{n}},\mathbb{R}_{\mathsf{n}} \rangle$ *designate a design model that subsumes* m. *To conclude our proof we will show that* $\overset{\ni}{\phi}$ *is true in* n.

By Definition VII, $p_1,\ldots p_n$, $f_1,\ldots f_n$, *and* $m_1,\ldots m_n$ *exist also in* $\mathbb{U}_{\mathsf{n}}$. *Also,* $\mathbb{R}_{\mathsf{n}}$ *includes relations* $Member$ *and* $Produce$ *that are supersets of the respective relations in* $\mathbb{R}_{\mathsf{m}}$. *Thus, the same triplets satisfy the relations* $Member$ *and* $Produce$ *in* $\mathbb{R}_{\mathsf{n}}$. *Thus,* $\overset{\ni}{\phi}$ *is true in* n.

?

Less formally, we say that Factory Method is local because if one design models incorporates an instance of the pattern then any design model that subsumes it also contains the atoms and relations that constitute the same instance, i.e., the same instance of ϕ?

The same line of reasoning can be applied to the specifications [9] of most of the design patterns from the Gamma et. al [15] catalogue.

5.2 Other Design Specifications

The place of other design specifications within the intension/locality classification may be less obvious then that of design patterns. Thus, we have carried out our analysis on the formal rendering of two additional design specifications: MVC (Model-View-Controller) "usage pattern" in Java™ *Swing* class library, and of Enterprise JavaBeans™.

In lack of space, we cannot quote here the formal specifications but only the results of our analysis. The interested reader may find both specifications in [10]. We can report that our analysis confirms that, as predicted by the intension/locality thesis, both specifications fall under the "design" category, namely, they are intensional and local.

6. Architectural Specifications

In this section, we demonstrate that two of the most common architectural styles are both intensional and non-local, fitting with our definition of what it means to be "architectural", from section 1.1. We also demonstrate that the Law of Demeter is, in fact, architectural.

6.1 Layered Architecture

Garlan and Shaw [17] describe the *layered architecture* style such that "An element of layer *k* may depend only on elements of layers *1,…k*." Formally, this description can be rephrased in $\mathbb{Z}$ as follows:

$$\forall e\ \exists! k \in \{\mathbb{N}\} \bullet Layer(e) = k \qquad (3.1)$$

(i.e., each element is defined in exactly one layer), and

$$\forall x,y \bullet \quad (3.2)$$
$$Depends(x,y) \Rightarrow Layer(x) \geq Layer(y)$$

(i.e., the definition of each element may "depend" only on the definition of elements of same layer or of lower layers.)

It is trivial to prove that conjunction of formulas (3.1) and (3.2) (designated α below) can be satisfied by an unbounded number of programs. Thus, α is intensional.

Corollary 4. *"Layered architecture" is non-local.*

Proof: *Let* $\mathfrak{m} = \langle \mathbb{U}, \mathbb{R} \rangle$ *designate a design model such that* $\mathfrak{m} \models \alpha$. *It is easy to construct a new model* $\overline{\mathfrak{m}} = \langle \overline{\mathbb{U}}, \overline{\mathbb{R}} \rangle$ *that subsumes* $\mathfrak{m}$ *(namely,* $\mathfrak{m} \preceq \overline{\mathfrak{m}}$*), but* α *is not true therein (namely,* $\overline{\mathfrak{m}} \not\models \alpha$*), as follows:*

- $\overline{\mathbb{U}} \triangleq \mathbb{U} \cup \{\xi\}$, *where* ξ *is an entity that is not in* $\mathbb{U}$*;*
- $\overline{\mathbb{R}} \triangleq \mathbb{R}$

Clearly, $\overline{\mathfrak{m}}$ *subsumes* $\mathfrak{m}$. *As* $Layer(\xi)$ *is not defined, expression (3.1) is not true in* $\overline{\mathfrak{m}}$ *and* $\overline{\mathfrak{m}} \not\models \alpha$. *Thus,* α *is non-local.*

?

Less formally, to prove this corollary we simply show that, to any program that is layered, it is easy to add an element that violates the principle.

Of course, we may selectively apply a non-local specification φ only to certain parts of a program. But what does it mean if only part of our program is "layered"? Does it mean that *Layered architecture* can sometimes be local?

Discussion. Is it possible for a non-local specification, such as layered architecture, be restricted only to a part of the program? Obviously. That simply means that parts of the program satisfy the non-local rule, while other parts violate it. In fact, this property is exactly what makes the layered architecture non-local: *Because* it may be violated anywhere in the program, we say that is not localized to a specific portion of the system. This example is illustrated in Figure 2 and in the discussion which follows.

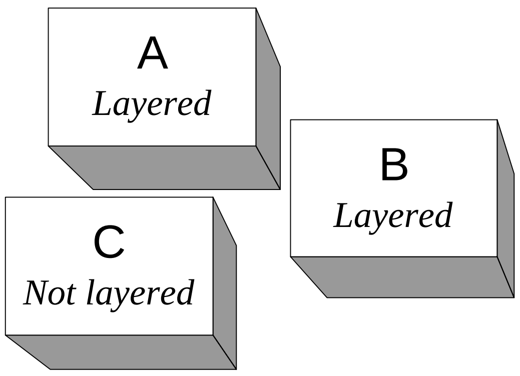


Figure 2. A three-module, partially-layered program

1.	Is *layered architecture* (expressions (3.1) and (3.2)) non-local?	*True* (always, by definition)
2.	Does this program satisfy the *layered-architecture*?	*False*
3.	Does part of this program satisfy *layered architecture*?	*True*

Does it mean that non-locality is meaningless? Not at all. Non-locality is a property of a *specification*; and, as we further demonstrate in the following subsections, architectural specifications are always non-local. However, we may choose to apply a specification only to some parts of the system, and violate it in others.

To summarize, it is not meaningless neither contradictory to state that architectural specifications can be applied selectively such that they are violated by some parts of a specific program. Formally:

***Corollary** 5. If a specification φ is (deliberately) violated in module m in program π, then either one of the following is true:*

- *φ is not satisfied by π, or*
- *m is not part of π (i.e., it belongs to a separate program.)*

In conclusion, if an architectural style is violated by even one part of the program, it can no longer be considered an architectural property of this program.

In the example of *layered architecture*, a module that does “layer bridging” in a layered architecture program (i.e., violates the layering principle) should not be considered as part of the layered program; instead, we perceive it a separate program.

While this conclusion may seem counter-intuitive at first, it is actually a powerful view on exceptions to architectural constraints. A module that does layer bridging requires different reasoning and different management than the rest of the layered system. It not only should, but *must*, be treated as an exception, or else the power of the layering will be compromised. Exceptions to an architectural style should have attention called to them and be made the focus of intense analysis. Our reasoning provides a sound, formal basis for saying when a portion of a program is an exception to an architectural style.

6.2 Pipes and Filters

According to Garlan and Shaw [17], ”In a pipes and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs.”

Dean and Cordy [7] present a visual formalism defined as a context-free grammar, and formulate the pipes and filter style as follows:

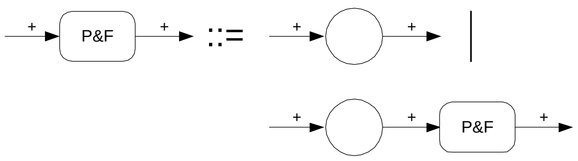


Figure 3. Pipes and Filters (adapted from [7]). A circle in the visual language represents a “task”, arrows represent streams. The plus sign is the BNF symbol for “one or more.”

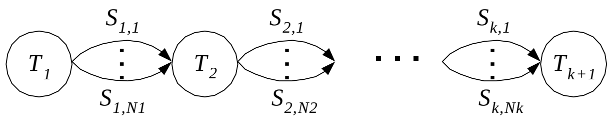


Figure 4. The general structure of programs that parse Figure 3.

A “program” in STSA (in absence of an explicit name, the authors’ initials serve us with reference to the formalism) is represented as a typed, directed multigraph. An *architectural style* is defined as a context-free language. Thus, an expression in STSA defines a collection of graphs in a visual notation, such as Figure 3. Hence, according to Dean and Cordy, a “program” satisfies the pipes and filters architecture if it “parses” Figure 3. Figure 4 illustrates the kind of programs that parse the grammar defined in Figure 3.

Before we can reason about Figure 3, we must demonstrate a denotation function which maps every “program” in STSA to a design model. For the purpose of our discussion in this section, it suffices to restrict ourselves to multigraphs that contain tasks and streams. More specifically, let *G* designate a multigraph whose nodes are “tasks” and whose arcs are “streams”. In the denotation we choose, tasks and streams are mapped into atoms. The relations we have in our respective design model consist of

- the unary relations $Task(x)$ and $Stream(x)$, and
- the binary relation $Connect(x,y)$

which indicates that the directed arc representing stream *x* terminates at the node representing task *y* (or that the directed arc representing stream *y* begins at the node representing task *x*.)

It is easy to see that the general form of programs that parses Figure 3 is that of a directed multi-path, as depicted in Figure 4, and that the design models of these programs have the general form illustrated in formula (4).

$$\begin{array}{l} Connect(T_1,S_{1,1}),\ldots Connect(T_1,S_{1,N1}),\ \ldots \\ \quad Connect(S_{1,1},T_2),\ldots Connect(S_{1,N1},T_2), \\ \ldots \\ Connect(T_k,S_{k,1}),\ \ldots Connect(T_k,S_{k,Nk}),\ \ldots \\ \quad Connect(S_{k,1},T_{k+1}),\ldots Connect(S_{k,Nk},T_{k+1}) \end{array} \tag{4}$$

It is trivial to see that there is an unbounded number of such programs. Therefore, the architectural style Pipes and Filters (denoted ϕ) is intensional.

***Corollary** 6. “Pipes and Filters” is non-local.*

***Proof:** Let m designate a model that satisfies ϕ. It is easy to construct a model that subsumes m but does not satisfy ϕ: By simply adding a new task to the model, T_x, such* *that no stream connects to T_x, we get a new model that subsumes* m *and yet does not satisfy* π. *Therefore* ϕ *is not local.*

?

6.3 Law of Demeter

So far we have shown that two classic architectural styles meet our criteria for being "architectural". This is expected. But our criteria also turn up some less expected results: The *Law of Demeter* [26] was created as a *design* heuristic. It was introduced to simplify modifications to a program and to reduce its complexity. The informal description of the law for functions is given in Table 4.

Table 4. Law of Demeter for functions

For all classes *C*, and for all methods *M* attached to *C*, all objects to which *M* sends a message must be instances of classes associated with the following classes: ▪ The argument classes of *M* (including *C*). ▪ The instance variable classes of *C*. (Objects created by *M*, or by functions or methods which *M* calls, and objects in global variables, are considered arguments of *M*.)

We may formulate the language of Table 4 as follows:

$$\begin{aligned} &\forall\ f_1,f_2,c_1,c_2 \bullet \qquad (5)\\ &\quad Member(f_1,c_1) \wedge\\ &\quad Member(f_2,c_2) \wedge\\ &\quad Invoke(f_1,f_2) \Rightarrow\\ &\quad Member(c_1,c_2) \vee ArgOf(c_2,f_1) \vee c_1=c_2 \end{aligned}$$

Formula (5) is evidently intensional and non-local. That is, it has infinitely many instantiations and it pervades the entire program. To prove this, observe that any program that satisfies (5) can be expanded with source code that violates the law of Demeter, such as demonstrated by the C++ source code in Table 5. In fact, it is the *violation* that proves that the law of Demeter is non-local.

Table 5. An add-on to a C++ program which violates the Law of Demeter

```
struct NewName1 {
   void foo();
};
struct NewName2 {
   NewName1 y;
};
class NewName3 {
   NewName2 x;
   void bar() {
      x.y.foo();
   }
};
```

In conclusion, the law of Demeter, which was created as a design rule, is in fact architectural, i.e., it is not expected to be limited to one part of the system but satisfied throughout, in coding practices and in design walk-throughs.

This example demonstrates the benefit of making our distinctions explicit and the power of rendering them precise, without which we would be unable to determine this property (of the Law of Demeter) conclusively.

7. Analysis

Clearly, the case studies in Sections 5 and 5.2 are not coincidental. The same line of reasoning used for the Factory Method can be used for many other (if not all) of the design patterns in [15], as well as for the architectural styles by Garlan and Shaw [17]. Examples drawn from other formal languages proposed for the specification of design patterns, such as *Constraint Diagrams* [25], *DisCo* [29], and *Contracts* [21], bring forth sample specifications, are clearly *intensional* as well. This motivates the following hypothesis:

The ***hypothesis of intensional specifications****. All "design patterns" and "architectural styles" are intensional.*

A direct proof for this hypothesis requires a formalization of "all" design patterns and architectural styles. The first problem with this is that no given catalogue purports to contain "all" patterns and styles, nor do we expect such a catalogue to be possible (except perhaps in the analytic sense.) Another problem arises from the mostly informal definitions given to patterns and styles. Limited attempts have been made to prove this hypothesis [14] [11], but naturally the proofs provided do not cover all known patterns and styles. That is why the "hypothesis of intensional specifications" remains a hypothesis.

7.1 Specific "Design" and "Architectures"

With the increase in popularity of the terms and their proliferation in the literature, the terms *design* and *architecture* often appear in a concrete context, as in "the design of this program." This usage implies that these terms can also be used with reference to extensional specifications, but only with reference to a concrete program. We suggest that "the design of program *x*" refers to the *instance* implemented in a program of a general design rule (e.g., design pattern.) Since instances (Definition II) are extensions, this resolves the apparent difficulty in the intension/locality thesis.

7.2 "Extensional" Vs. "Local"

It appears that non-locality is a "stronger" property than intensionality. For example, we expect every program to be local. Below, we prove this intuition.

Definition IX*. The sequence of* ***natural extensions*** *to a design model* $\mathsf{m} = \langle \mathbb{U}_{\mathsf{m}}, \mathbb{R}_{\mathsf{m}} \rangle$ *is the infinite sequence of models* $\mathsf{m}_0, \mathsf{m}_1, \mathsf{m}_2, \ldots$ *such that:*

- $\mathsf{m}_0 \triangleq \mathsf{m}$
- $\mathsf{m}_i \triangleq \langle \mathbb{U}_i, \mathbb{R} \rangle$ *and* $\mathbb{U}_i = \mathbb{U}_{i-1} \cup \{e_i\}$

where $e_1, e_2 \ldots$ *is an infinite sequence of "new" entities that are pairwise distinct and none of which is in* $\mathbb{U}_{\mathsf{m}}$.

Corollary 7*. It is trivial to show that each design model in the sequence of natural extensions to* m *subsumes it.*

Theorem*. Non-local specifications are also intensional.*

Proof: *We prove this by showing that if a specification is extensional then it is local. Let* φ *designate an extensional specification. Let* $\mathsf{m} = \langle \mathbb{U}_{\mathsf{m}}, \mathbb{R}_{\mathsf{m}} \rangle$ *designate a design model such that* $\mathsf{m} \models \varphi$*. It suffices to show that there exists a design model* n *such that* $\mathsf{m} \preceq \mathsf{n}$ *but* $\mathsf{n} \not\models \varphi$*.*

From Corollary 1 we conclude that the set of design models where φ *is true is finite, while the set of natural extensions (Definition IX) to* m*:* $\mathsf{m}_0, \mathsf{m}_1, \mathsf{m}_2, \ldots$ *is infinite. Therefore, there exists a model* n *in the sequence of natural extensions to* m *such that* $\mathsf{n} \not\models \varphi$*. By Definition IX,* $\mathsf{m} \preceq \mathsf{n}$*.*

?

We may conclude that the relation between extensional, intensional, and non-local specifications can be summarized in the following Venn diagram:

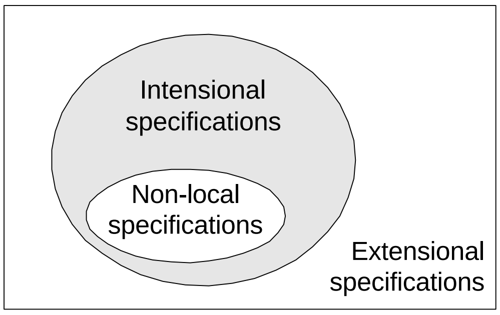


Figure 5. Intensional vs. non-local specifications

7.3 UML Class Diagrams

Since UML is used widely as a design and architectural notation, it is of particular interest to understand the place of UML diagrams in the classification we introduced: Are they local? Intensional?

Despite the wide-spread attempts towards rendering the notation with well-defined semantics (e.g., the research group known as *pUML* [35]), most types of UML diagrams have no well-defined semantics. Thus, our discussion therein is largely informal, assuming that any formal interpretation for class diagrams will be consistent with the informal semantics.

In terms of design models, we can assume that any such interpretation will associate class icons with atoms of type *class*, operations with atoms of type *method*, as well as provide us with a specification of a set of associated relations. It is easy to see why the specification given by a class diagram is local; but is it intensional?

To answer this question, observe that a UML class diagram is commonly viewed as an under-specification; that is, it is a specification whose implementations may have any number of additional elements that are not mentioned in the diagram. Under this assumption, UML class diagrams are intensional, since there is an unbounded number of elements that can be added to any implementation.

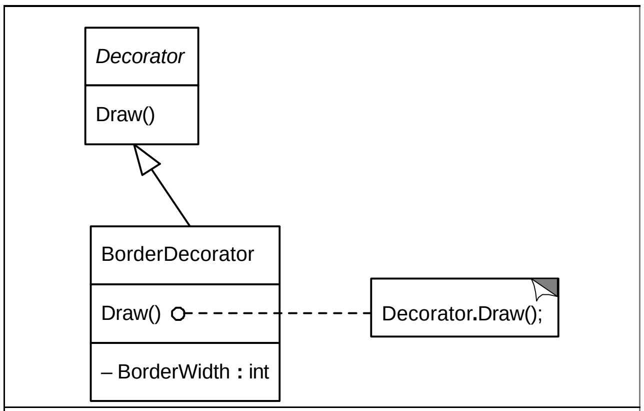
Unlike the formulas used in our examples, however, the abstraction that class diagrams provide is of the most rudimentary type, that is, by omitting information but without using free variables. A UML diagram provides no information on the elements that are not explicitly described. Thus, class diagrams are intensional only in a trivial sense, in the same way that the code excerpts in Table 2 and Table 3, if taken not as complete programs but just as excerpts thereof, are intensional. Clearly, this sense is quite unlike the way architectural and design specifications are intensional. The following definition facilitates this distinction:

Definition X*. An intensional specification* φ *is* ***quasi-extensional*** *iff* $[\![\varphi]\!]$*, namely, the set of design models that satisfy* φ*, has a single lower bound with respect to the partial-ordering relation "subsumption".*

It is trivial to show that subsumption induces partial ordering on a set of design models. Definition X, however, assigns specific importance to sets of design models that contain one member such that all other design models subsume it.

Corollary 8*. UML class diagrams are quasi-extensional.*

Figure 6 illustrates why UML class diagrams are quasi-extensional. In contrast, it is trivial to show that the intensional specifications quoted in this paper are not quasi-extensional.



The minimal design model that satisfies this diagram appears in Table 2. Since every program that implements this diagram satisfies it, there are infinitely-many other design models that satisfy it.

Figure 6. A UML class diagram and its interpretations

8. Summary

The terms *architecture*, *design*, and *implementation* have been heavily used by both researchers and practitioners for over a decade, but this usage has always been informal and intuitive. But this intuition has not always provided a sound basis for reasoning, discussion, and documentation. This paper provided a sound formal basis for the distinction between the three terms, based upon best practices.

The intension/locality thesis argues that architectural styles are *intensional* and *non-local*, design patterns are *intensional* and *local*, and implementations are *extensional* and *local*. We provided a proof for parts of our thesis and demonstrated in detail the truth underlying the remainder.

What are the consequences of precisely knowing the differences between the terms architecture, design and implementation? The ramifications are many. Among others, these distinctions facilitate –

- the distinction between *architecture* and *design* documents;
- the distinction between local and non-local rules, i.e., between the design rules that need be enforced throughout a project vs. those that are of a more limited domain;
- determining what constitutes a uniform program, e.g., a collection of modules that satisfy the same architectural specifications.

For example, in the industrial practice of software architecture, many statements that are said to be 'architectural' are in fact local, e.g., *both tasks A and B execute on the same node*, or *task A controls B*. Instead, a truly Architectural statement would be, for instance, *for each tasks A,B which satisfy* φ, *A and B will execute on the same node and* $Control(A,B)$. More generally, for each specification we know we should determine whether it is a *design* statement, describing a purely local phenomenon (and hence of secondary interest in documentation, discussion, or analysis), or whether it is this an instance of an underlying, more general rule.([2])

Acknowledgements

Many thanks to Jens Jahnke and Alejandro Allievi for their comments. We thank Mary U. Anna for her inspiration. This research was supported in part by the Natural Sciences and Engineering Research Council, Canada and by the U.S. Department of Defense.

References

[1] J. Barwise, ed., (1977). *Handbook of Mathematical Logic*. Amsterdam: North-Holland Publishing Co.

[2] L. Bass, P. Clements, R. Kazman (1998). *Software Architecture in Practice.* Reading, MA: Addison Wesley Longman, Inc.

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal (1996). Pattern-Oriented Software Architecture – A System of Patterns. New York, NY: Wiley and Sons.

[5] S. A. Cook, R. A. Reckhow (1973). “Time-Bounded Random Access Machines.” *Journal of Computer and System Sciences*, Vol. 7, pp. 354—475.

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

[7] T. R. Dean, J. R. Cordy. "A Syntactic Theory of Software Architecture." *IEEE Trans. on Software Engineering* 21 (4), Apr. 1995, pp. 302—313.

[8] F. DeRemer, H. H. Kron. "Programming-in-the-large versus programming-in-the-small." *IEEE Trans. in Software Engineering* 2 (2), June 1976, pp. 80—86.

[9] A. H. Eden. “Formal Specification of Object-Oriented Design.” *International Conference on Multidisciplinary Design in Engineering CSME-MDE 2001*, Nov. 21—22, 2001, Montreal, Canada.

[10] A. H. Eden. "LePUS: A Visual Formalism for Object-Oriented Architectures." *The 6th World Conference on Integrated Design and Process Technology*, Pasadena, California, June 26—30, 2002.

2 This final example was suggested by an anonymous ICSE reviewer.

[11] A. H. Eden. "A Theory of Object-Oriented Design." *Information Systems Frontiers* 4 (4), Nov.—Dec. 2002. Kluwer Academic Publishers.

[12] A. H. Eden, Y. Hirshfeld. "Principles in Formal Specification of Object Oriented Architectures." *CASCON 2001*, Nov. 5—8, 2001, Toronto, Canada.

[13] A. H. Eden, R. Kazman (2003). "On the Definitions of Architecture, Design, and Implementation". Technical report, Department of Computer Science, University of Essex.

[14] P. van Emde Boas (1997). "Resistance Is Futile; Formal Linguistic Observations on Design Patterns." Research Report no. CT-19997-03, The Institute for Logic, Language, and Computation, Universiteit van Amsterdam.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

[16] D. Garlan, R. Monroe, D. Wile (1997). "ACME: An Architectural Description Interchange Language." *Proceedings of CASCON'97*. Toronto, Ontario.

[17] D. Garlan, M. Shaw (1993). "An Introduction to Software Architecture." In V. Ambriola, G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, Vol. 2, pp. 1—39. New Jersey: World Scientific Publishing Company.

[18] J. Gosling, B. Joy, G. Bracha (2000). *The Java™ Language Specification*, 2nd edition. Reading, MA: Addison Wesley Longman, Inc.

[19] Y. Gurevich. "Sequential Abstract State Machines Capture Sequential Algorithms". *ACM Trans. on Computational Logic* 1 (1), July 2000, pp. 77—111.

[20] D. Harel. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8 (3), June 1987, pp. 231—274.

[21] R. Helm, I. M. Holland, D. Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." *Proceedings OPPSLA/ECOOP*, Oct. 21—25, 1990, Ottawa, Canada.

[22] I. Kant (1781). "The Critique of Pure Reason". Translated into English by N. K. Smith, Macmillan Press Ltd.

[23] R. Kazman (1999). "A New Approach to Designing and Analyzing Object-Oriented Software Architecture." Guest talk, *Conference On Object-Oriented Programming Systems, Languages And Applications – OOPSLA'99*.

[24] L. Lamport (1994). "The Temporal Logic of Actions." *ACM Trans. on Programming Languages and Systems* 16 (3), May 1994, pp. 872—923.

[25] A. Lauder, S. Kent (1998). "Precise Visual Specification of Design Patterns." In *Proceedings of the 12th ECOOP, Brussels, Belgium*, July 1998. Lecture Notes in Computer Science 1445. E. Jul (ed.) Berlin: Springer-Verlag.

[26] K. Lieberherr, I. Holland, A. Riel (1988). "Object-oriented programming: an objective sense of style." *Conference proceedings OOPLA'88*. San Diego, CA, pp. 323—334.

[27] D. C. Luckham et. al. "Specification and Analysis of System Architecture Using Rapide." *IEEE Trans. on Software Engineering* 21 (4), Apr. 1995, pp. 336—355.

[28] B. Meyer (1991). *Eiffel: The Language*. New Jersey, NJ: Prentice Hall.

[29] T. Mikkonen (1998). "Formalizing Design Patterns." *Proceedings of the International Conference on Software Engineering*, April 19—25, 1998, pp. 115—124. Kyoto, Japan.

[30] R. T. Monroe, A. Kompanek, R. Melton, D. Garlan. "Architectural Styles, Design Patterns, and Objects." *IEEE Software* 14(1), Jan. 1997, pp. 43—52.

[31] D. E. Perry, A. L. Wolf (1992). "Foundation for the Study of Software Architecture." ACM SIGSOFT *Software Engineering Notes*, 17 (4), pp. 40—52.

[32] C. A. Petri (1962). "Communications with Automata." *Technical report RADC-TR-65-377*. Princeton, NJ, Applied Data Research.

[33] Popkin Software (2000). *System Architect 2001*. New York, NY: McGraw-Hill.

[34] R. Prieto-Diaz, J. Neighbors. "Module Interconnection Languages." *Journal of Systems and Software* 6 (4), 1986, pp. 307—334.

[35] *The Unambiguous UML Consortium* page: `www.cs.york.ac.uk/puml/`

[36] T. Quatrani (1999). *Visual Modelling with Rational Rose 2000 and UML, Revised*. Reading, MA: Addison Wesley Longman, Inc.

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

[38] SEI (2002). Carnegie Mellon's *Software Engineering Institute*. `http://www.sei.cmu.edu`.

[39] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.

[40] J. M. Spivey (1989). *The Z Notation: A Reference Manual*. New Jersey: Prentice Hall.

[41] B. Stroustrup (1997). *The C++ Programming Language*, 3rd edition. Reading, MA: Addison-Wesley.

[42] A. Taivalsaari (1996). "On the Notion of Inheritance." *ACM Computing Surveys*, Vol. 28, No. 3, pp. 438—479.

[43] A. Turing (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* 2 (42), 1936-7, pp. 230—236.

[44] J. M. Vlissides, J. O. Coplien, N. L. Kerth (1996). *Pattern Languages in Program Design 2*. Reading, MA: Addison-Wesley.