

*A version of this paper has been accepted for publication in: International  
Conference on Software Engineering — ICSE, May 3–10, 2003, Portland, OR.*

defines “architectural patterns” that are in par with what the “Gang of Four” [14] call “design patterns”.

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 [30], and it has become the industry de facto standard ADL, although UML was specifically designed 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*® [28] and *System Architect*® [25]. Using the same specification language further blurs the distinction between artifacts of the design (class diagrams) from the implementation (source code.)

**Intended contribution.** Why are we interested in such distinctions? Naturally, a well-defined language improves our understanding of the subject matter. 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 formal ontology we provide can serve as the ultimate reference point for these discussions.

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.

Many of the required definitions were rendered informal and the proofs were omitted in this version of our work. The interested reader can find the complete definitions in [12], and additional details on the *design models* formalism can be found in [9]. Instead, we argue our points with informal illustrations and discussions.

## 1.1 The Intension/Locality Thesis

To elucidate the relationship between *architecture*, *design*, and *implementation*, we distinguish at least two separate interpretations for *abstraction* in our context:

- *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;
- *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 contribute to the distinction among *architecture*, *design*, and *implementation*, summarized 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*.

**Table 1.** The Intension/Locality Thesis

Architecture	<i>Intensional</i>	<i>Non-local</i>
Design	<i>Intensional</i>	<i>Local</i>
Implementation	<i>Extensional</i>	<i>Local</i>

## 1.2 Structure of This Paper

The intension/locality thesis can be understood correctly only in the context of the ontology provided below. In Section 2 we define *design models*, which are crucial to the remainder of our discussion. Design models are abstractions which allow a formal “meaning” assigned to programs, also called “denotation”. This formalism allows 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 *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 in 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 *Enterprise 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 of UML class diagrams in Section 7.2 reveals that 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 illustrate the formal ontology that underlies our discussion. This ontology is based on giving an abstract “meaning” to programs using *design models*.

### 2.1 Design models

*Turing* and *random-access machines* provide robust computational models that are primarily suitable for reasoning about algorithms. Other computational models and formalisms (e.g., Petri nets, statecharts, and temporal logic) facilitate reasoning about certain behavioral properties.

The discussion in architectural and design specifications, however, involves reasoning on constructs such as *classes*, *methods*, and *function calls*. Most other formalisms incorporate too much 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 different formalism, one which abstracts programs to a more convenient representation.

Eden and Hirshfeld [11] demonstrate how to model source code as *design models*, which are first order, finite structures in mathematical logic [1]. We designate a **design**

**Table 2.** A Java™ program and its denotation (adapted from [9].)

<pre> <b>abstract class</b> Decorator {   <b>public void</b> Draw(); } <b>class</b> BorderDecorator <b>extends</b> Decorator {   <b>public void</b> Draw() {     Decorator.Draw();   }   <b>private int</b> BorderWidth; } </pre>
<p>The design model of this program consists of the following:</p> <p><u>Atoms:</u></p> <p>“class” atoms: {Decorator, BorderDecorator, int, void}, “Method” atoms: {BorderDecorator.Draw, Decorator.Draw }</p> <p><u>Relations:</u></p> <p><i>Abstract</i>(Decorator)</p> <p><i>Member</i>(Decorator.Draw, Decorator)</p> <p><i>Member</i>(BorderDecorator.Draw, BorderDecorator)</p> <p><i>Inherit</i>(BorderDecorator, Decorator)</p> <p><i>Reference</i>(BorderDecorator, int)</p> <p><i>Invoke</i>(BorderDecorator.Draw, Decorator.Draw)</p> <p><i>ReturnType</i>(Decorator.Draw, void)</p> <p><i>ReturnType</i>(BorderDecorator.Draw, void)</p>

**model**  $m$  (defined formally in [11] and [12]) to consist of a set of *atoms* and a set of *relations* among those atoms.

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 to a collection of *classes* and *methods* (also *routines* or *function members*) and their relations. *Atoms* represent *classes* and *methods* that were defined in the program, such as the class `Decorator` and the method `Decorator.Draw`. *Relations* represent their correlations, such as

$$\textit{Inherit}(\text{BorderDecorator}, \text{Decorator}) \quad (1)$$

Note that *design models* are abstractions which were made to reflect certain structural aspects of computer programs that are relevant to the discussion in software design theory. Obviously, this representation limits the type of reasoning we may perform to properties that are relevant to the discussion in classes, methods, and their interdependencies, as opposed to the discussion in dynamic properties such as *fairness* and complexity.

### 2.2 Specifications and instances

In this subsection, we discuss specifications and programs, and how the two correlate. We make some reasonable assumptions on the languages used to write specifications.

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

*SPEC* includes familiar specification languages such as  $\mathbb{Z}$  [32], as demonstrated in formulas (5.1) and (5.2), and LePUS [8], as demonstrated in formula (4). *SPEC* also includes programming languages such as Eiffel, ANSI C, C++, and Java™. Naturally *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 wish to ask: Does this program satisfy our specification? To answer this question, consider for example the following trivial design specification:

$$\textit{Inherit}(x, y) \quad (2)$$

Expression (2) contains two free variables  $x, y$ . We say that it can be *satisfied* by any pair of atoms that belong to the relation *Inherit*. For example, from expression (1) we conclude that the pair  $\langle \text{BorderDecorator}, \text{Decorator} \rangle$  satisfies expression (2).

Thus, we say that the pair of atoms  $\langle \text{BorderDecorator}, \text{Decorator} \rangle$  is an **instance** of (2), and that the design model depicted in Table 2 **satisfies** ex-

pression (2). A more formal definition of *instance* appears in [12] and in [9].

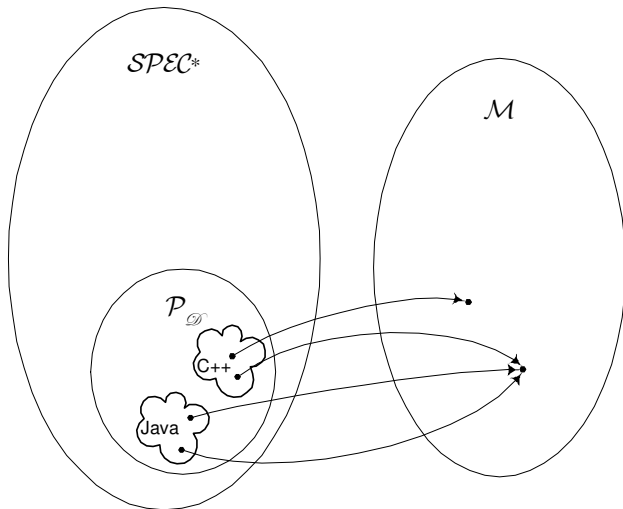
From this example, it should be clear that an *instance* is not the same as a “program”. Depending on the specification, a program may contain zero, one, or any number of instances. The following subsection formalizes the notion of a program.

### 2.3 Programs and their denotation

We expect a “program” to be a specification that is associated with only one *design model*. The association between “real” programs and design models is provided by the *denotation function*. Loosely speaking, a **denotation function**  $\mathcal{D}$  has these properties:

- Its domain, denoted  $\mathcal{P}_{\mathcal{D}}$ , is a subset of  $SPEC^*$ ;
- $\mathcal{D}$  associates each element  $\varphi$  with *exactly one* design model (its *denotation*) which *satisfies*  $\varphi$ .

Typically, the domain of  $\mathcal{D}$  contains expressions in programming languages, such as C++ and Eiffel. More formally, a **program** is an element in  $\mathcal{P}_{\mathcal{D}}$ . Every design model typically denotes infinitely many programs. Figure 1 illustrates a typical denotation associating programs with design models. Table 2 illustrates the denotation of a simple Java™ program and Table 3 the denotation of a C++ program.



**Figure 1.** A typical denotation function

Observe that the set of all programs,  $\mathcal{P}_{\mathcal{D}}$ , is only a *small subset* of the expressions in  $SPEC^*$  and that there are many expressions in  $SPEC^*$  that are satisfied by more than one design model

We now have a well-defined notion of programs and specifications. In combination with the definition of an “in-

stance” of a specification, we can conclusively determine whether a program satisfies a given specification:

**Definition I.** We say that a program  $\pi$  *satisfies* specification  $\varphi$  iff the design model of  $\pi$  satisfies  $\varphi$ .

This means that, for example, the Java™ program depicted in Table 2 *satisfies* expression (2).

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*, which will be applied, both formally and informally, to selected specifications in the following sections.

### 3.1 The Intension criterion

Perry and Wolf [24] 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, the North Atlantic Treaty defines the set of NATO members extensionally, namely, by itemizing its members: United States, United Kingdom, Norway, and so forth.

The notion of intensionality defined below 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.** We say that a specification is *intensional* iff there are infinitely-many possible instances thereof. Conversely, all other expressions are *extensional*.

It immediately follows (as shown in [12]) that intensional specifications are also satisfied by infinitely many design models and by infinitely many programs.

### 3.2 The Locality criterion

Monroe et. al [23] argue that “Architectural designs are typically concerned with the entire system.” Similarly, we observe that an architectural style, which pervades a system [16], manifests a property that is shared across modules of the system. This intuition motivates the Locality criterion: What distinguishes architectural from design specifications is that *architectural specifications must be met by every extension of the program*.

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

$$\forall c \in \mathbb{C} \bullet (c \neq \text{Object}) \Rightarrow \text{Inherit}^*(c, \text{Object}) \quad (3)$$

(Where *Inherit*\* is the transitive closure of the binary relation *Inherit*.) The Intension/Locality thesis argues that Universal Base Class is architectural because (a) it is intensional and (b) it pervades *all parts* of the system, i.e., *every* class must be bound to *Object*.

**Subsumption.** The formalization of the locality criterion requires the notion of *subsumption*. Informally, we say that design model *n* *subsumes* *m* if *m* is a “sub-model” of *n*. We can also view *n* as an “extension” to *m*, not unlike “strict inheritance” [33].

**Definition III.** We say that a specification  $\varphi$  is *local* iff the following condition holds:

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

Essentially, Definition III states that an expression is *local* if it can be satisfied in “some corner” of our program without this being affected in how the rest of the program is like.

In the following sections, we apply the Intension and Locality criteria to selected specifications to illustrate the difference between programs, design specifications, and architectural specifications.

## 4. Implementations

It immediately follows from the Intension criterion (as shown in [12]) that intensional specifications are also satisfied by infinitely many design models and by infinitely many programs. This leads us to prove part (iii) of the Intension/Locality thesis:

**The lemma of “extensions”:** *Programs are extensional specifications.*

Following the ‘principle of least constraint’, we expect architectural specifications to have an unbounded number of instances, namely, to be “intensional”. The same applies to design patterns. But what about other forms of specifications? Can programs be intensional?

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 demonstrated in Corollary 1, specifications in any programming language, including generics and interpreted code are, under the assumptions provided above, purely extensional:

**Corollary 1.** *C++ templates are extensional.*

To illustrate this, consider the design model of a C++ program with templates, such as shown in Table 3.

**Table 3.** A C++ program and its denotation

<pre> <b>template</b> &lt;<b>class</b> C&gt; <b>class</b> Stack   { /* ... */ <b>int</b> main() {   Stack&lt;<b>int</b>&gt; si;   <b>return</b> 0; } </pre>
<p>This program is interpreted by only one design model, which consists of the following:</p> <p><b>Atoms:</b>  “Class” atoms: {Stack, si} "Method" atoms: {main}</p> <p><b>Relations:</b>  <i>Generic</i>(Stack)  <i>Instantiate</i>(Stack, si, int)  <i>Return</i>(main, int)</p>

Corollary 1 demonstrates that although generics may be viewed as intensional with respect to other semantic frameworks, e.g., because they can be used to define other concrete constructs, the ontology we have provided assigns then with only one “interpretation”. The reason is that the formal semantics we chose for the representation of programs are *design models*, which are more abstract than the machine code generated from compilation and from other formal frameworks.

To recap, the formal framework provided in Section 2 guarantees that expressions in all conventional programming languages are extensional.

## 5. Design specifications

By part (ii) of the Intension/Locality thesis, design specifications should be local and intensional. Since design specifications are, in practice, defined informally, we begin section with exploring the intuition behind our thesis. For the purpose of the formal analysis which follows, however, we make use of formal specifications and apply them to widely recognized designs.

### 5.1 Design patterns

In this subsection, we focus on an example drawn from the published patterns literature. This allows us to test our ideas on some of the most widely used design specifications.

Coplien and Schmidt [5] 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” [29], 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*.

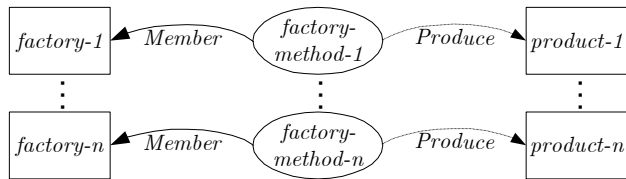
Consider, for example, the Factory Method design pattern [14]. Essentially, the pattern’s solution offers three sets of participants:

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



**Figure 2.** The general structure of the “Factory Method” pattern

For the discussion in design patterns we use LePUS, a formal specification language for object-oriented design which is defined in detail in [8]. The five statements in the above definition of the Factory Method are formalized by expressions (4.1) to (4.5), respectively:

$$\text{Products} : \mathbf{P}(\mathbb{C}) \quad (4.1)$$

$$\text{Factories} : \mathbf{P}(\mathbb{C}) \quad (4.2)$$

$$\text{FactoryMethods} : \mathbf{P}(\mathbb{F}) \quad (4.3)$$

$$\text{Clan}(\text{FactoryMethods}, \text{Factories}) \quad (4.4)$$

$$\text{Produce}^{\leftrightarrow}(\text{FactoryMethods}, \text{Products}) \quad (4.5)$$

Expressions (4.1) through (4.3) declare two sets of *classes* and one set of *methods*. Expression (4.4) indicates that the pair  $\langle \text{FactoryMethods}, \text{Factories} \rangle$  satisfies the predicate *Clan*, meaning that –

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

Finally, expression (4.5) indicates that the ground relation *Produce* is a bijective function between the set *FactoryMethods* and the set *Products*.

**Corollary 2.** “Factory method” is *intensional* and *local*.

The composite expression (4.1) to (4.5) is evidently *intensional*, since each one of the free variables *Products* and *Factories* (*FactoryMethods*) can be instantiated by any number of classes (methods). To show that it is *local*, observe that if a design model  $\mathfrak{m}$  satisfies expression (4) then it incorporates an *instance* of the pattern. It is then easy to show that any proper “extension” to  $\mathfrak{m}$  (i.e., any design model that *subsumes* it) also contains the same instance of the Factory Method, namely, the same set of atoms and relations that satisfied the pattern in  $\mathfrak{m}$ .

The same line of reasoning can be applied to the specifications of most of the design patterns from the Gamma et al [14] catalogue, such as the specifications in [8].

## 5.2 Other design specifications

The place of other design specifications within the intension/locality classification may be less obvious than 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], and the complete proof for this conclusion in [12]. 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, as predicted by the intension/locality thesis, two classic architectural styles are both intensional and non-local. We also demonstrate that the Law of Demeter is architectural.

### 6.1 Layered architecture

Garlan and Shaw [16] describe the Layered Architecture such that “An element of layer  $k$  may depend only on elements of layers  $1, \dots, k$ .” We may formalize this description in  $\mathbb{Z}$  as follows:

$$\forall e \exists! k \in \{\mathbb{N}\} \bullet \text{Layer}(e) = k \quad (5.1)$$

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

$$\forall x, y \bullet \quad (5.2)$$

$$\text{Depends}(x, y) \Rightarrow \text{Layer}(x) \geq \text{Layer}(y)$$

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

**Corollary 3.** *Layered Architecture is intensional and non-local.*

It is obvious that an unbounded number of programs can satisfy expression (5) (i.e., the conjunction of formulas (5.1) and (5.2)), hence it is intensional. To prove that it is non-local we show that, given any design model that satisfies (5), the same design model can be extended with a new element in the lower “layer” such that this element depends on a higher layer, thereby violating the specification.

We conclude that we may selectively apply a non-local specification only to certain parts of a program. But what does it mean? Does it compromise our results?

**Discussion.** Is it possible for a non-local specification, such as layered architecture, to be applied only by one part of a program? Obviously. That simply means that parts of this program satisfy the non-local rule, while other parts violate it. In fact, this property is exactly what the Locality criterion is about: We say that the Layered Architecture specification is non-local *because* it may be violated anywhere in the program.

Does it mean that non-locality is inconsequential? Quite the contrary. Non-locality is a “strong” property that qualifies *specifications*, i.e., of an expression; and by our thesis, architectural specifications are always non-local. However, they may be selectively violated in any part of the program as a conscious choice of the architect. In such case we say that this architectural style no longer characterizes this program (as a whole.) Formally:

**Corollary 4.** *If a specification  $\varphi$  is (deliberately) violated in module  $m$  of program  $\pi$ , then either one of the following is true:*

- $\varphi$  is not satisfied by  $\pi$ , or
- $m$  is not considered part of  $\pi$  (i.e., it belongs to a separate 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.

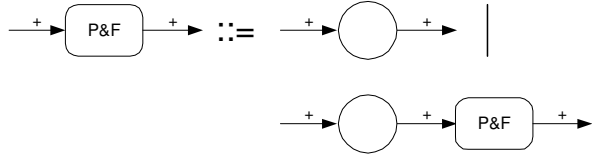
While this conclusion may seem counter-intuitive at first, it is actually a powerful view on exceptions to architectural constraints. A module that does layer bridging requires different reasoning and different management than the rest of the layered system. It not only should, but *must*, be treated 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

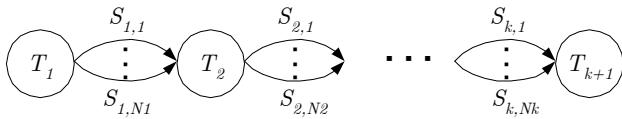
According to Garlan and Shaw [16], “In a pipes and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs.”

Dean and Cordy [6] present a visual formalism defined as a context-free grammar, and formulate the pipes and filter style as depicted in Figure 3. More generally, a “pro-

gram” in STSA (\*) 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.



**Figure 3.** Pipes and Filters (adapted from [6]). Circles represent tasks, 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.

Before we can reason about Figure 3, we must establish which design models satisfy an STSA diagram. For the purpose of our discussion, it suffices to restrict ourselves to diagrams of “tasks” and “streams”. Let  $G$  designate a multigraph whose nodes represent “tasks” and whose arcs represent “streams”. In terms of design models, tasks and streams are represented as atoms, while the relations consist of –

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

which indicate 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 parse 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 (6).

$$\begin{aligned}
 &Connect(T_1, S_{1,1}), \dots, Connect(T_1, S_{1,N1}), \dots & (6) \\
 &Connect(S_{1,1}, T_2), \dots, Connect(S_{1,N1}, T_2), \\
 &\dots \\
 &Connect(T_k, S_{k,1}), \dots, Connect(T_k, S_{k,Nk}), \dots \\
 &Connect(S_{k,1}, T_{k+1}), \dots, Connect(S_{k,Nk}, T_{k+1})
 \end{aligned}$$

**Corollary 5.** “Pipes and Filters” is intensional and non-local.

It is obvious that an unbounded number of programs satisfy formula (6); therefore, it is intensional. To show that (6) is non-local, observe that, for any design model that satisfies (6) we can add a new task  $T_{k+2}$  that is not connected to any other task by a pipe. Such addition violates the style’s specification (in particular, there would be no  $Connect$  relation involving  $T_{k+2}$ ). Since violates (6) *whenever* it occurs, the Pipes and Filters style must be non-local.

### 6.3 Law of Demeter

We have shown that two classic architectural styles meet our criteria for being “architectural” as expected. But our criteria also turn up some less expected results: The *Law of Demeter* [20] 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:

$$\begin{aligned}
 &\forall f_1, f_2, c_1, c_2 \bullet & (7) \\
 &Member(f_1, c_1) \wedge \\
 &Member(f_2, c_2) \wedge \\
 &Invoke(f_1, f_2) \Rightarrow \\
 &Member(c_2, c_1) \vee ArgOf(f_1, c_2) \vee c_1 = c_2
 \end{aligned}$$

Evidently, formula (7) has infinitely many instances, hence it is intensional. It is also non-local. To prove this, observe that any program that satisfies (7) can be expanded with source code that violates the Law, such as demonstrated by the C++ source code in Table 5.

\* In absence of an explicit name, we use the initials of the title of [6] with reference to the formalism.

**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 better characterized as an *architectural* rule. The Law is not be limited to one part of the system but must be satisfied throughout. In practice, this means that any system using the Law of Demeter must create appropriate architectural practices to enforce it via coding standards, design walkthroughs, tool support, etc.

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

## 7. Analysis

Clearly, results obtained from the case studies in Sections 5 and 6 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 [14], as well as for the architectural styles by Garlan and Shaw [16]. Examples drawn from other formal languages proposed for the specification of design patterns, such as *Constraint Diagrams* [19], *DisCo* [22], and *Contracts* [17], 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 would require 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 (e.g.,[13],[9]), but naturally the proofs provided cannot cover all known patterns and styles. Fr this reason, our hypothesis remains as such.

## 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 are extensions, this resolves the apparent difficulty in the intension/locality thesis.

## 7.2 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* [27]), most types of UML diagrams have no well-defined semantics. Thus, our discussion here 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, note that a UML class diagram is commonly viewed as an under-specification, namely, an incomplete specification; the actual implementation of the diagram may have any number of additional elements that are not mentioned in the diagram at all. Under this assumption, UML class diagrams are intensional, since there exists an unbounded number of elements that can be added to any implementation.

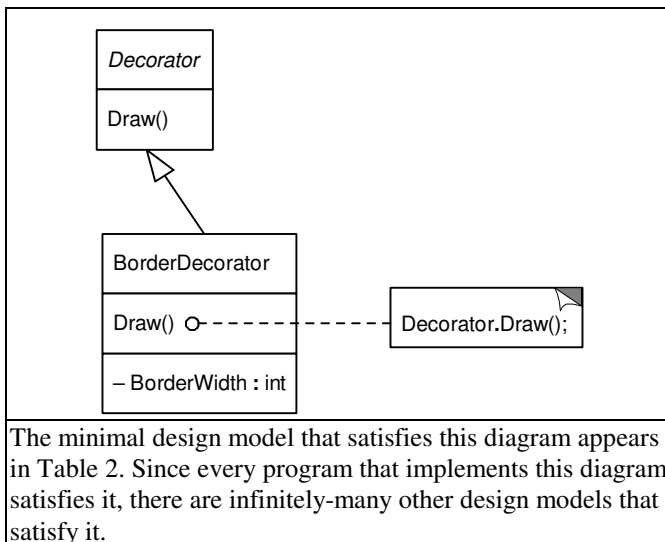
Unlike the formulas used in our examples, however, the abstraction that class diagrams provide is of the most rudimentary type, that is, by omitting information but without using free variables. A UML diagram provides no information on the elements that are not explicitly described. Thus, class diagrams 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 IV.** An intensional specification  $\varphi$  is **quasi-extensional** iff the set of design models that satisfy  $\varphi$ , has a single lower bound with respect to the partial-ordering relation “subsumption”.

It is easy to show that subsumption induces partial ordering on a set of design models. Definition IV, however, assigns specific importance to sets of design models that contain design model such that all other members subsume it.

**Corollary 6.** UML class diagrams are quasi-extensional.

Figure 5 illustrates the proof to this corollary. It is trivial to show also that none of the intensional specifications we quoted above is quasi-extensional.



**Figure 5.** A UML class diagram and its interpretations

## 8. Conclusions

We have provided a sound formal basis for the distinction between the terms *architecture*, *design*, and *implementation*, called the Intension/Locality thesis, and established it upon best practices.

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

- determining what constitutes a uniform program, e.g., a collection of modules that satisfy the same architectural specifications;
- determining what information goes into *architecture* documents and what goes into *design* documents;
- determining what to examine and what to not examine in an architectural evaluation or a design walkthrough;

- understanding the distinction between local and non-local rules, i.e., between the design rules that need be enforced throughout a project versus those that are of a more limited domain.

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 some property  $\varphi$ , 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. (\*)

## Acknowledgements

We than Lutz Prechelt of the ICSE program committee for his very detailed comments. Many thanks to Jens Jahnke and Alejandro Allievi for their contributions. We thank Mary J. 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] J. Coplien, D. Schmidt, eds. (1995). *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.
- [6] T. R. Dean, J. R. Cordy. "A Syntactic Theory of Software Architecture." *IEEE Trans. on Software Engineering* 21 (4), Apr. 1995, pp. 302–313.
- [7] 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.
- [8] 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.
- [9] A. H. Eden. "A Theory of Object-Oriented Design." *Information Systems Frontiers* 4 (4), Nov.—Dec. 2002. Kluwer Academic Publishers.

\* This final example was suggested by an anonymous ICSE reviewer.

- [10] A. H. Eden. "LePUS: A Visual Formalism for Object-Oriented Architectures." *The 6th World Conference on Integrated Design and Process Technology*, Pasadena, CA, June 26—30, 2002.
- [11] A. H. Eden, Y. Hirshfeld. "Principles in Formal Specification of Object Oriented Architectures." *CASCON 2001*, Nov. 5—8, 2001, Toronto, Canada.
- [12] A. H. Eden, R. Kazman (2003). "On the Definitions of Architecture, Design, and Implementation". Technical report CSM-377, January 2003, Department of Computer Science, University of Essex, United Kingdom.
- [13] 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.
- [14] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
- [15] D. Garlan, R. Monroe, D. Wile (1997). "ACME: An Architectural Description Interchange Language." *Proceedings of CASCON'97*. Toronto, Ontario.
- [16] 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.
- [17] R. Helm, I. M. Holland, D. Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." *Proceedings OPPSLA/ECOOP*, Oct. 21—25, 1990, Ottawa, Canada.
- [18] R. Kazman. "A New Approach to Designing and Analyzing Object-Oriented Software Architecture." Guest talk, *Conference On Object-Oriented Programming Systems, Languages and Applications – OOPSLA*, Nov. 1—5, 1999, Denver, CO.
- [19] A. Lauder, S. Kent. "Precise Visual Specification of Design Patterns." *Proceedings of the 12th ECOOP, Brussels, Belgium*, July 1998. LNCS 1445. Berlin: Springer-Verlag.
- [20] 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.
- [21] 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.
- [22] T. Mikkonen (1998). "Formalizing Design Patterns." *Proceedings of the International Conference on Software Engineering*, April 19—25, 1998, pp. 115—124. Kyoto, Japan.
- [23] R. T. Monroe, A. Kompanek, R. Melton, D. Garlan. "Architectural Styles, Design Patterns, and Objects." *IEEE Software* 14(1), Jan. 1997, pp. 43—52.
- [24] D. E. Perry, A. L. Wolf (1992). "Foundation for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes* 17 (4), pp. 40—52.
- [25] Popkin Software (2000). *System Architect 2001*. New York, NY: McGraw-Hill.
- [26] R. Prieto-Diaz, J. Neighbors. "Module Interconnection Languages." *J. of Systems and Software* 6 (4), 1986, pp. 307—334.
- [27] *The Unambiguous UML Consortium* page: [www.cs.york.ac.uk/puml/](http://www.cs.york.ac.uk/puml/)
- [28] T. Quatrani (1999). *Visual Modelling with Rational Rose 2000 and UML, Revised*. Reading, MA: Addison Wesley Longman, Inc.
- [29] 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.
- [30] SEI (2002). Carnegie Mellon's *Software Engineering Institute*. <http://www.sei.cmu.edu>.
- [31] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.
- [32] J. M. Spivey (1989). *The Z Notation: A Reference Manual*. New Jersey: Prentice Hall.
- [33] A. Taivalsaari (1996). "On the Notion of Inheritance." *ACM Computing Surveys* 28 (3), pp. 438—479.