Published: Hawaii International Conference on System Sciences — HICSS-38, January 3-6, 2005, Honolulu, HI

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

STRATEGIC VERSUS TACTICAL DESIGN

Amnon H. Eden

Department of Computer Science, University of Essex, United Kingdom
and
Center For Inquiry International, Amherst, NY

Abstract. *We seek to distinguish* Strategic *design decisions (e.g., to adopt a programming paradigm, architectural style, CBSE standard or application framework) from* tactical *design decisions (e.g. to use a design pattern, refactoring or programming idiom). This distinction is important since strategic statements carry far-reaching implications over the implementation and therefore must be made early in the development process, whereas tactical statements have localized effect and must be deferred to a latter stage in the process.*

We formulate the ***Locality criterion****, a well-defined and language-independent criterion which divides all design statements in two abstraction classes. We apply our criterion to a broad range of statements and demonstrate that strategic statements are non-local and that tactical design statements are local. We also demonstrate that assumptions leading to architectural mismatch are non-local.*

1 Introduction

Strategic design decisions determine the primary behavioural and structural properties of the system under development. Strategic decisions include the choice of programming paradigm, architectural style, application framework, component-based software engineering (CBSE) standard and global design principles, depending on the application domain. Strategic design decisions address general, system-wide concerns and set up the context to the entire implementation. Implementations rarely conform to more than one strategic statement and attempting to combine components that assume different design strategies may result in *architectural mismatch* [11]. Because they carry the most consequential implications, strategic decisions must be considered early in the development process and established explicitly before any other design decisions are made.

In contrast, tactical design decisions have limited ("local") effect on the software under development and therefore can (and should) be taken much later in the process. For the most part, tactical design aims to overcome shortcomings of the programming language, paradigm or the operating environment. Tactical statements include lower-level abstractions, such as design patterns, refactorings and programming idioms, and they are often described in terms of specific mechanisms of the programming language, such as specific objects and procedures.

Architectural vs. detailed design

The distinction between strategic vs. tactical may benefit from examining definitions of *software architecture*. Kazman states that "Architecture, or architectural design, is design at a higher level of abstraction." [16] Perry and Wolf write in the seminal paper on the foundations for software architecture: "*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." [19] Thus, it may be argued that the distinction between strategic vs. tactical corresponds to the distinction made between architectural vs. detailed design.

While the concept of architectural design is generally understood, there is however no common agreement on its precise boundaries. Compare for instance the Publisher-Subscriber [1] architectural pattern with the Observer design pattern [10]: Although both patterns evidently describe the same class of implementations, the first was classified as "architectural pattern" while the other as a "design pattern".

In [7] we present the Locality criterion (revised in [6]), which divides design statements into two *abstraction classes*: local statements (designated $\mathscr{L}$) vs. non-local statements (designated $\mathscr{NL}$). In [6] we also formulate the **Intension/Locality Hypothesis** which, *inter alia*, argues the following:

- Architectural statements are *non-local*
- Design statements are *local*

Overview

In this paper, we expand the Intension/Locality Hypothesis in favour of a broader claim: We argue that the Locality criterion, defined and illustrated in Section 2 below, distinguishes between *any* strategic design decision from *any* tactical design decisions.

To demonstrate this argument, we examine in Section 3 a broad range of design statements and demonstrate that strategic statements are in $\mathscr{NL}$ and that tactical statements are in $\mathscr{L}$.

We also argue that the locality criterion can aid us in the understanding the reasons for architectural mismatch. *Architectural mismatch* [11] has been defined as the class of interoperability problems arising from conflicting assumptions a component makes about the structure and behaviour of the application in which it is intended to operate. We argue that the class of assumptions leading to architectural mismatch is a subclass of $\mathscr{NL}$. In Section 4, we examine the causes of architectural mismatch and demonstrate that these are non-local statements.

In Section 5 we summarize the arguments presented in this article and draw conclusions therefrom.

Intended contributions

Our theory has both conceptual and practical implications. First and foremost, we see significant interest in providing a well-organized structure to software design, one which is not unlike complexity classes in computational complexity. We are unaware of any other rigorous framework that provides a similarly uniform, panoramic view on design statements, and believe that the distinction between L and NL is a first step in this direction. This distinction is designed to capture the intuitive notion of non-locality using a coherent, concise and well-defined criterion.

We provide a common reference ontology for the discussion in abstraction classes of design statements. Since the Locality criterion is a semantic rather than syntactic criterion, we can apply it to a broad range of design statements articulated in variety formal, semi-formal and informal, textual and visual languages, including (but not restricted to) first- and high-order predicate calculus (**PC**), context-free languages, Z [4] and LePUS. This allows us to compare a broad spectrum of phenomena which otherwise may seem incommensurable. In this sense, our theory departs from existing literature.

Finally, the Locality criterion explains why tools supporting non-local design statements must examine the entire implementation. For example, compilers enforcing Information Hiding (Statement (3)) must inspect the entire implementation to ensure that private members are not accessed by any part of the implementation.

But the results we provide also have practical implications: The Locality criterion distinguishes between the subjects of the *Architecture* vs. the *Design* documents, which most large software development projects require. Our analysis leads to the conclusion that *architectural mismatch* (Section 4) can be minimized if we observe that interoperability is predicated upon the consistency between non-local assumptions made by the components we attempt to combine. In Figure 10 we detail how the Locality criterion contributes to the understanding of three of the "four necessary aspects of a long-term solution" to the problem of architectural mismatch.

2 Abstraction classes

In [6] we establish a common reference ontology which allows us to compare statements about the software design in formal, semi-formal and informal languages, specified either in a textual or in a visual form. Stated in simple terms of entities and relations, this vocabulary will allow us to examine a range of strategic and tactical design statements and to determine their abstraction class. In this section, we recap on this vocabulary, define the locality criterion and illustrate its application. In lack of space, we omit the formal definitions, most of which can be found in [6].

2.1 Implementations and their semantics

Which notion of program semantics best suits our discussion? Eden and Hirshfeld [5] provide abstract semantics for programs in the form of first-order, finite structures in mathematical logic [1]. By this approach, each implementation is implicitly accompanied by a mathematical structure that consists of a universe of entities (such as $class$ or $component$) and relations (such as $Inherit$ or $InModule$). A straightforward representation to the vocabulary of entities and relation can be formulated using the notion of a *structure*. Formally, we define **design model** as a finite collection of (ground) *entities* and (ground) *relations* between the entities. Figure 2 illustrates a design model, $[\![Nil]\!]$, which represents the abstract semantic of the C++ program in Figure 1.

A design model can also be viewed as a relational database, consisting of a tabular representation for each relation. By this metaphor, $[\![Nil]\!]$ can be viewed as a miniature database with three tables: The table $Class$ has one column with the entries `Object`, `Nil1` and `Nil2`. The two-column tables $Inherit$ and $Members$ have one entry each, (`Nil1`,`Object`).

```
class Object { /* ... */ };

class Nil1: public Object {
  vector<Object*> Objs;
};

class Nil2 { /* ... */ };
```

Figure 1. Nil, a 'toy' C++ program.

Entities: `Object`, `Nil1`, `Nil2`
Relations:
- $Class = \{$`Object`, `Nil1`, `Nil2`$\}$ (unary relation)
- $Inherit = \{($`Nil1`,`Object`$)\}$ (binary relation)
- $Members = \{($`Nil1`,`Object`$)\}$ (binary relation)

Figure 2. $[\![Nil]\!]$, abstract semantics of the toy C++ program in Figure 1.

We assume the existence of a denotation function $\mathscr{D}$ which maps every implementation p to a design model, $[\![p]\!]_{\mathscr{D}}$. We will assume such a fixed function throughout our discussion. Thus, we may omit the designation of the denotation function from the representation of the design model of p, which shall be written simply as $[\![p]\!]$. The details of a denotation function are not relevant to our discussion. For a detailed discussion in abstract interpretation functions, see [7].

In the discussion that follows, we demonstrate the use design models to determine the abstraction class of a range of design statements.

2.2 The Locality criterion

The Locality criterion originally appeared in [7]. The criterion was revised in [6]. The difference between the versions is very subtle and does not affect our discussion. Below, we give the revised version and refer to it simply as the Locality criterion:

The Locality Criterion. A statement φ is *local* if and only if it is preserved under *expansion*.

We designate the class of local design statements $\mathscr{L}$, non-local design statements $\mathscr{NL}$. We term each class of statements as **abstraction class**. ([1])

([1]) The term "class" in this context is taken from Zermelo-Fraenkel's set-theoretic vocabulary, designating an extension of a property. We also discuss object-oriented and class-based programming languages, in which context the term "class" refers to the grammatical construct which defines the structure and behaviour of *instances* (also *objects*). These notions of "class" must not be confused.

Informally, a statement is *local* if and only if, once it is satisfied by some part of the program, it cannot be violated by "expanding" the program.

The notion of *expansion* must be treated with care. Not any string of text added to the source code or variable added to the memory model is considered an expansion. An expansion can only add new entities, it may not modify existing entities. Thus, to determine which entities can be added in an expansion, we must ask: What type of entities can be added to the existing implementation without directly modifying it?

Example: non–local statement

The principle of a **Universal Base Class** may reflect a design decision applied to a particular context, for example, in stating that the inheritance tree of the NIHCL C++ class library ([2]) has a single root. It can also be taken as a statement describing every possible program in a certain programming language, such as Java™, Smalltalk or Eiffel. The principle can be formulated in the **PC** as follows:

$$\forall c \bullet Class(c) \Rightarrow Inherit^*(c, root) \qquad (1)$$

where $Inherit^*$ is short for the transitive closure (zero or more) of the binary relation $Inherit$.

Let us demonstrate informally that Statement (1) is not preserved under expansion: We show a program that satisfies the statement and an expansion to this program that does not satisfy the statement:

1 The universe of discourse consists of class definitions in the C++ programming language and the inheritance relation(s) between them.
2 Let p_0 designate the C++ program that includes the classes `Nil1` and `Object` (excluding class `Nil2`) and the relation $Inherit$. Clearly, p_0 satisfies Statement (1).
3 Let p designate the C++ program that results from adding class `Nil2` to p_0. Clearly, p is an expansion to p_0, but p does not satisfy Statement (1).

A more formal proof is detailed in [6].

Example: local statement

The "intent" of the **Recursive Composite** design pattern [10] is to "Compose[s] objects into tree structures to represent part-whole hierarchies". Essentially, the technique described involves two "participants":

([2]) NIHCL: A class library for C++ from the US National Institutes of Health (NIH).

- An abstract $component$ class, which declares the interface for objects in the composition and for accessing and managing its child components;
- A concrete $composite$ class, which stores "children" objects and defines behaviour related to managing children.

Below, we formulate this description in the **PC**:

$$\begin{aligned} &Class(component) \wedge \\ &Class(composite) \wedge \\ &Inherit(composite, component) \wedge \\ &Members(composite, component) \end{aligned} \qquad (2)$$

Statement (2) is local. Let us sketch an informal argument to this claim in the context of the C++ programming language:

1 The universe of discourse consists of (object-oriented) class definitions in the C++ programming language and the inheritance relations between them.
2 Let p_0 be the C++ program illustrated in Figure 1. To show that p_0 satisfies Statement (2), replace the free variable $component$ with the entity `Object`, and the variable $composite$ with the entity `Nil1`.
3 Clearly, this assignment remains true in any expansion to p_0. Hence, Statement (2) is preserved under expansion.

Although the argument given above is made for C++ implementations of the Recursive Composite, note that a similar argument can be made about implementations in other programming languages

2.3 Caveat

The application of mathematical logic and model theory to statements about software design raises several problems. First, most design statements are specified informally, and therefore the vocabulary we require (specifically, the language of *structures* and *expansions*) is not rigidly defined. We partially overcome this problem by "borrowing" the formulations of design statements whenever such formulations appeared in literature. When formal definitions could not be found, informal statements in natural language were formulated using the classical **PC**. But as we also demonstrate in [6], the Locality criterion can be equally applied to informal statements, as long as it has a definite meaning within the vocabulary defined.

The next problem we faced was that even the sporadic formulations of design statements were made in a plethora of specification languages. How can we compare statements made in Z with statements in a context-free language? We overcome this problem by phrasing a criterion that is based on the meaning of statements rather than on their form. In a confusing reality of multiple notations, we hope that a semantic criterion is more informative than syntactic characterizations. The bewildering variety of notations and languages of the examples given in Section 3 were deliberately chosen to demonstrate this quality.

We use mathematical logic to ensure that the arguments we make are valid and to promote accuracy and clarity. In this paper, we avoided unnecessarily obfuscated or elaborate discussion and adhere to the informal argumentation. Rigorous arguments and complete proofs can be found in [6].

More importantly, "proving" our hypothesis is a daunting task: The technical specifications of programming paradigms, application frameworks and CBSE standards are much more elaborate than the simplified formulas we quote and therefore rarely lend themselves to set-theoretic analysis. Also, the terms *strategic* and *tactical* are not rigorously defined, and neither is almost any of the technical terms we analyze (such as *architectural style* and *design pattern*). Consequently, our hypothesis cannot be "proved" but merely corroborated (or contradicted) empirically. But most difficult is to justify the inductive leap we attempt to perform: What right do we have to generalize a finite set of corroborating evidence into a sweeping hypothesis?

The credence of every scientific thesis is predicated upon fine-combing the class of phenomena it is concerned with. But the subject matter of our hypothesis, the class of design statements and their interpretations, is very broad, intricate and largely undefined. It cannot be covered in any one paper. Instead, we demonstrate how to determine the abstraction class of any design statement that can be expressed in the vocabulary we define. We hope that this vocabulary is sufficiently general and that the insight it provides is sufficiently useful to merit further interest.

2.4 Discussion

Can we express the Locality criterion syntactically? An alternative to the semantic approach we presented could operate within the confines of one specification language, thereby allowing the formulation of the criterion by syntactic terms. Turner [24], for example, describes a core theory of specifications which is comprehensive enough to express *all* the statements that we are interested in. This rich formal theory will eventually shed light on the practical use of specification languages in discussing programs, including Z, VDM and LePUS. In particular, his approach rigorously *represents* the features of the languages used in dealing with programs and provides means for investigating the logical implications of these features. This approach, however, remains to be examined.

3 Case studies

In this section, we examine a range of design statements and demonstrate that strategic statements are non-local and that tactical statements are local.

3.1 Strategic design

Programming paradigms

The choice of a programming paradigm can easily be recognized as strategic: It determines the principal abstractions underlying the implementation (e.g., objects, procedures, or functions) and the primary mechanisms that the programming language offers to support them.

The definition of a programming paradigm goes beyond the scope of this paper. Consider for example the principle of **Information Hiding**, which is supported by access restrictions in modular and in object-oriented programming. In the C++, Java™ and Eiffel programming languages, the principle is supported by a mechanism which renders a 'private' member accessible only to a privileged part of the program and inaccessible to the rest. This rule can be formulated in the **PC** as follows:

$$\forall c,m,x \bullet \quad (3)$$
$$Member(m,c) \wedge Private(m) \wedge Access(x,m) \Rightarrow$$
$$Public(m) \vee Member(x,m) \vee Friend(x,m)$$

To demonstrate that Statement (3) is non-local, we show a program that satisfies it, illustrated in Figure 3, and then expand this program into one that violates the statement, such as by adding the class `Intruder` illustrated in Figure 4.

Since the C++ compiler enforces Statement (3), attempting to compile the combined program will lead to a compilation error. This demonstrates why tools that enforce non-local rules must examine the entire implementation. In this example, the compiler must ensure that the access privileges are respected by all parts of the program. This result also demonstrates the actual implications of non-locality.

```
template <class T> class Stack {
public:
    void push(T);
    // ...
private:
    T * theStack;
    int size;
};

Stack<complex<float> > si;
```

Figure 3. Class `Stack` in C++.

```
class Intruder {
    int foo() {    return si.size; }
};
```

Figure 4. An expansion to the program in Figure 3 that violates Information Hiding.

Global design principles

Design principles are "global" design decisions that potentially affect every module of the program. For example, the design principle of a Universal Base Class (Statement (1)), can reflect a design decision made in a particular context, for example, in stating that the inheritance tree of the NIHCL C++ class library has a single root. This principle is clearly strategic since it implies that all objects (also *instances*) will exhibit the set of properties defined in the root class.

Universal Base Class also describes the structure of any program in some programming language. For example, any class declaration written in Java™, Smalltalk or Eiffel inherits (by definition) from class `Object`. In Section 2.2 we showed that Universal Base class is non-local.

Architectural styles

Architectural styles [19][12] (also architectural patterns [22]) have emerged as common means for specifying the design principles underlying the global organization of complex systems. Architectural styles are concerned with inter-process communication protocols, physical distribution of modules and components, calling conventions, security concerns, global control structures and performance goals. Clearly, architectural styles are design statements at the highest level of abstraction and characteristically strategic.

Consider for example the **Implicit Invocation** architectural style, described in Figure 5. Essentially, the style restricts inter-module communication to the use of events. Let us illustrate why this statement is non-local:

Let $\mathfrak{M}_{ii}$ designate a structure that satisfies Figure 5, whose universe consists of $Module$ elements $M=m_1,...m_k$ and of $Procedure$ elements $P=p_1,...p_n$.

> The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

Figure 5. Implicit invocation architectural style [12].

The relations in $\mathfrak{M}_{ii}$ include the binary relations $InModule \subset P \times M$ and $Invoke \subset P \times P$. $\mathfrak{M}_{ii}$ can be expanded by adding a two modules m_{k+1}, m_{k+2} and two procedures p_{n+1}, p_{n+1}, and by expanding the relations $InModule$ and $Invoke$ such that –

- $InModule(p_{n+1}, m_{k+1})$
- $InModule(p_{n+2}, m_{k+2})$
- $Invoke(p_{n+1}, p_{n+2})$

This would constitute a proper expansion because it does not modify any existing module or procedure in $\mathfrak{M}_{ii}$. Clearly, the expanded structure violates the constraints imposed by the style. Thus, Implicit Invocation is not preserved under expansion. QED

Consider also the **Layered Architecture** style. Garlan and Shaw [12] state that "A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below." In [7] we prove that a formulation of this statement in the **PC** is non-local.

Finally, consider the **Pipes and Filters** architectural style, described in [12] as follows: "In a pipes and filters 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." As we prove in [7] with regard to a formulation of this statement in a visual, context-free language, it is non-local.

Component-based software engineering

CBSE industrial standards determine components interaction, integration, customization, calling and usage conventions [23]. Depending on the standard, CBSE conventions allow the interoperability of binaries across process, machine, operating system and programming language boundaries. Each CBSE standard sets up the context to all the components in the implementation. Evidently, the choice of a CBSE standard is a strategic design decision.

Each CBSE standard consists of a complex constellation of constraints. An detailed discussion in the technical specifications of any CBSE standards is beyond the scope of this paper. Instead, we discuss segments from the description of two of these standards.

Microsoft's **Component Object Model** (COM) requires each component to implement the interface `IUnknown`. This statement is a slight variation on a special case of Universal Base Class (Statement (1)), which is non-local by the same proof given in Section 2.2.

Consider also the description of a "bean" class in **Enterprise JavaBeans**™ depicted in Figure 6. In [6] we formulate this statement in LePUS and prove that it is non-local.

- Every bean obtains an `EJBContext` object, which is a reference directly to the container.
- The bean class defines create methods that match methods in the home interface and business methods that match methods in the remote interface.

Figure 6. EJB statements (adapted from [17]).

Application frameworks

Johnson describes an application framework as "a reusable, 'semi-complete' application that can be specialized to produce custom applications." Application framework usually provide substantial parts of the overall application – most often extensive resources and the infrastructure for the implementation. The choice of framework usually affects the control structure (i.e., due to the principle of inverted control), the choice of programming language, and most often the gross organization of the application. We conclude that the choice to use an application framework is clearly a strategic design decision.

For example, **Microsoft Foundation Classes** (MFC) is an application framework that supports the use of graphical user interface in Windows environments. Figure 7 depicts the interaction protocol between Microsoft Foundation Classes and user-defined classes. In [6] we prove that this description is non-local.

If users subclass `CWnd`, in all the subclasses, there must be at least one public method which directly or indirectly calls one of the three methods `CWnd::Create`, $\texttt{CWnd::CreateEx}_1$, or $\texttt{CWnd::CreateEx}_2$. Furthermore, outside the subclasses, there must be the invocation of at least one of the methods.

Figure 7. MFC interaction protocol (adapted from [13]).

3.2 Tactical design

Design patterns

In parallel with the emergence of architectural styles, design patterns were introduced as a vocabulary capturing best practices in software design. The seminal "Gang of Four" catalogue [10] defines design patterns as "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*". Unlike specifications of architectural styles, individual objects and minute implementation detail are common elements in specifications of design patterns. For this reason, Monroe et. al argue that design patterns "have applicability at lower levels of design" [18] compared to architectural styles. Similarly, Coplien refers in the same issue of *IEEE Software* to the solutions that design pat-

terns specify as "microarchitectures", echoing the view that design patterns manifest design decisions of lower abstraction level than architectural styles. In conclusion, the decision to adopt a design pattern is tactical.

In Section 2.2 we showed that a formulation of the **Publisher-Subscriber** [20] (called Observer in [10]) design pattern is local. Below, we demonstrate the same with regard to two other design patterns.

The **Factory Method** design pattern [10] describes three kinds of higher-order entities ("participants" in the catalogue's terminology), which can be represented as higher-order variables as follows:

1. A set of "product" classes, $Products$
2. A set of "factory" classes, $Factories$
3. A set of "factory" methods, $Factory\text{-}Methods$

The pattern also specifies the relations among these entities ("collaborations") as follows:

4. The set $Factory\text{-}Methods$ has the following properties:
   a. All the methods $m \in Factory\text{-}Methods$ are of the same "type".
   b. Each factory method $m \in Factory\text{-}Methods$ is defined in a different class in $f \in Factories$.
   c. Each factory method $m \in Factory\text{-}Methods$ is responsible for "producing" instances from exactly one class $p \in Products$. More precisely, we say that the relation $Produces$ is a bijection relation between the sets $Factory\text{-}Methods$ and $Products$.

In [7] we prove that a formulation of the factory method in LePUS is local. This proof demonstrates that once a collection of factories, factory methods and products that satisfies the statement is present in an implementation, it will also be present in any expansion thereof.

The intent of the **Strategy** design pattern [10] is to "define a family of algorithms, encapsulate each one, and make them interchangeable." [12] The design pattern describes a collection of "concrete strategy" classes and a family of "algorithm" methods defined therein. The interface of a "context" class provides the algorithms with the necessary data. We may formulate the Strategy design pattern using the Z schema depicted in Statement (4). Note that, unlike in classical interpretation for Z schemas, Statement (4) should be interpreted as an existential formula on the variables $context$, $strategy$, $ConcreteStrategies$, $algorithm$ and $ContextInterface$.

In [6] we prove that Statement (4) is local. Essentially, we show that once a collection of strategies and algorithms is present in an implementation, it will also be present in any expansion thereof.

$$
\begin{array}{l}
[CLASS] \\
[METHOD] \\
[SIGNATURE]
\end{array}
\qquad (4)
$$

$$
\begin{array}{l}
ANY : CLASS \cup METHOD \\
Member : ANY \leftrightarrow CLASS \\
Invoke : METHOD \leftrightarrow METHOD \\
\otimes : SIGNATURE \times CLASS \rightarrowtail\!\!\!\!\!\twoheadrightarrow METHOD
\end{array}
$$

Strategy

$$
\begin{array}{l}
context,\ strategy : CLASS \\
ConcreteStrategies : \mathbb{P}\ CLASS \\
algorithm : SIG \\
ContextInterface : \mathbb{P}\ SIGNATURE
\end{array}
$$

$$
\begin{array}{l}
(context, strategy) \in Member \\
\forall c \in ConcreteStrategies \bullet \\
\quad (c, strategy) \in Inherit^{+} \\
\forall c \in ConcreteStrategies \bullet \\
\quad \exists s \in ContextInterface \bullet \\
\quad (algorithm \otimes c, s \otimes Context) \in Invoke
\end{array}
$$

More generally, it is easy to see why design patterns are local. Each design pattern can be specified as a statement that assigns roles to bounded collections of 'participants (classes, objects and/or methods). Thus, any expansion to an implementation that satisfies the specification incorporates the same a collection of participants that also satisfies the same statement.

Programming idioms

Bushcmann et. al describe programming idioms as "low-level patterns" [2]. Programming idioms usually specify a technique for overcoming a particular shortcoming of implementation language, most often a mechanism that is directly supported by some other programming language. Therefore we conclude that the decision to use a programming idiom is tactical.

The **Counted Pointer** idiom [2] (originally termed Reference Counting by Stroustrup), summarized in Figure 8, describes a technique designed to overcome the absence of a "garbage collection" in C++. In [6] we prove that this statement is local.

1. The constructors and destructors of the Body class should be private.
2. The Handle class should be a "friend" of the Body class.
3. The body class should have a reference counter.
4. The Handle class should have a data member pointing to the Handle object.
5. The Handle class' copy constructor and assignment operator should increase the reference counter, and the destructor decrease it.
6. The Handle class should implement the arrow operator.
7. The Handle's constructors must initialize the reference counter with the number 1.

Figure 8. Counted Pointer programming idiom (adapted from).

Refactoring

Fowler defines refactoring as the process of "changing a software system in such a way that it does not alter the external behavior of the system." [9] An indication that the overall effect of refactoring is local is given in the catalogue: "[Refactorings]… take a bad design and rework it into well-designed code. … Yet the cumulative effect of these small changes can radically affect the design." Each refactoring has a small-scale effect and therefore the decision to apply one must be characterized as tactical.

Fowler's catalogue consists of a large number of refactoring. Some refactorings introduce a design pattern, such as Replace Type Code With Class (Strategy [10]), Replace Conditional With Polymorphism (State [10]) and Replace Constructor with Factory Method (Factory Method [10]). Other refactorings introduce a variable, a method or a new class; it is trivial to show that the statement describing the effect of these changes is local.

The same catalogue also describes 4 "big refactorings". Below, we demonstrate that one of them is local. The proof for the other three refactorings will become obvious from this example.

Tease Apart Inheritance [9] replaces one ("tangled") class hierarchy with another ("untangled"). In its general form, the refactoring suggests that one set of classes is replaced by two sets of classes. Thus, a statement describing this refactoring shall take the form of a well-formed formula in the **PC**, $\tau(h_1,h_2,h_3)$, where h_1,h_2 and h_3 are the only variables in the statement ranging over sets of class and τ describes their relations. To show that this statement is preserved under expansion, consider any implementation p that satisfies it, including three sets of classes `h1`, `h2` and `h3` such that τ(`h1,h2,h3`). Clearly, any expansion to p will incorporate the sets `h1`, `h2` and `h3` and will not affect their relations. Hence τ(`h1,h2,h3`) will remain true in any expansion of p and τ is local.

3.3 The Singleton anomaly

In absence of a formal criterion, the distinction between design patterns and architectural styles is open for interpretation. We argue that design patterns are tactical, hence local, whereas architectural styles are strategic, hence non-local. So far, we have only demonstrated limited evidence to our hypothesis. Our investigation of the remaining patterns in the "Gang of Four" catalogue [10], of design patterns in other collections [15][21] and of additional architectural styles [12], however, resulted in only one exception to our hypothesis: the Singleton pattern.

The intent of the Singleton design pattern [10] is to "Ensure a class only has one instance, and provide a global point of access to it." This statement is obviously non-local (see proof in [6]).

There are two possible explanations to this anomaly: (1) that design patterns can be non-local, and our hypothesis is wrong; (2) that the Singleton should be considered an architectural pattern/style.

When deciding which explanation applies, we ask: Is it a strategic decision to adopt the Singleton pattern? In illustrating the applicability of the Singleton pattern, Gamma et. al describe resources (printer spooler, file system and window manager) that are central to the successful operation of the overall system. The pattern is used to make these resources available to every element of the implementation. The term "global" used in the pattern's definition suggests that the singleton instance is designed to provide services to clients from the entire scope of the implementation. These evidence suggest that the decision on using the Singleton pattern is indeed strategic.

4 Architectural mismatch

"Many would argue that future breakthroughs in software productivity will depend on our ability to combine existing pieces of software to produce new applications". [11] Despite progress made since these words were written (in particular progress made in CBSE technologies), component integration and reuse have remained the panacea of software engineering.

In their seminal paper on architectural mismatch [11], Garlan et. al examine in detail the problems encountered during their attempt to construct a working system ("Aesop") from existing modules ("components"), such as excessive code size, poor performance and error-prone construction process. In the analysis of these problems, the authors define **architectural mismatch** as the class of interoperability problems arising from conflicting assumptions that modules make about the application in which

1. For example, the Softbench Broadcast Message Server expected all of the components to have a graphical user interface ….
2. packages made assumptions … about the data that would be communicated over the connectors. … Softbench, on the other hand, assumes that … all data to be communicated … is represented as ASCII strings. [However,] the main kind of data manipulated by our tools was database and C++ object pointers.
3. OBST … assumed that all of the tools would be completely independent of each other. This assumption meant that there would be no direct interactions between tools.
4. … the Mach Interface Generator assumed that the rest of the code was a flat collection of C procedures, and that its specification described the signature of all of these procedures.

Figure 9. Assumptions leading to architectural mismatch (adapted from [11]).

they are intended to operate. Clearly, the prevention of architectural mismatch is a strategic design concern.

Garlan et. al describe the particulars of several of the assumptions leading to architectural mismatch. Figure 9 gives a summary of some of these assumptions.

The authors do not provide a formal definition to architectural mismatch. Nonetheless, our examination of the assumptions leading to problems they describe (such as the statements in Figure 9) lead us to the conclusion that *architectural mismatch* may be defined in the language of structures and expansions as follows:

1. "*Make architectural assumptions explicit.*"	Make non-local assumptions explicit.
2. "*Construct large pieces of software using orthogonal components.* As Parnas has long argued [Par72], each module should hide certain design assumptions. Unfortunately, the architectural design assumptions of most systems are spread throughout the constituent modules. Ideally one would like to be able to tinker with the architectural assumptions of a reused system by substituting different modules for the ones already there."	Minimize the number of non-local assumptions. Ideally, each module should only make local assumptions.
4. *"Develop sources of architectural design guidance.* … We need to find ways codify and disseminate principles and rules for composition of software."	Design rules for composition: Whenever non-local assumptions are made, either ensure that they are compatible or else relinquish the attempt to combine the components.

Figure 10. Solutions to architectural mismatch (left, adapted from [11]) rephrased using non-locality (right).

Definition: Architectural Mismatch. Let φ designate a design statement assumed by component p. Let q designate an expansion of p. If $[\![p]\!]$ satisfies φ but $[\![q]\!]$ does not then we say that p, q *mismatch* on φ.

It is easy to prove that any assumption leading to *architectural mismatch* (in this formulation) is in $\mathcal{NL}$. Formally:

Theorem. If p, q *mismatch* on φ then φ is in $\mathcal{NL}$.

The proof to this theorem follows immediately from the Locality criterion.

In Figure 10 we demonstrate how the Locality criterion contributes to the understanding of three of the "four necessary aspects of a long-term solution" (to the problem of architectural mismatch).

5 Conclusions

We compared strategic with tactical design decisions and demonstrated that they can be distinguished using the Locality criterion. Specifically, we demonstrated that selected examples of programming paradigms, architectural styles, CBSE standards, design principles and application frameworks are non-local and that design patterns, refactorings and programming idioms are local.

From the examination of assumptions leading to architectural mismatch we conclude that they can be formulated as non-local statements. We also demonstrated that components will become more portable if they minimize the number of non-local assumptions they make about the applications in which they intend to operate.

Future directions

Further analysis may refine the abstraction classes $\mathcal{NL}$ and $\mathcal{L}$ ([3]). Also of interested would be the investigation of the abstraction class of metaprogramming statements, that is, of statements in a programming language that operate over programs.

Another open question is whether the class of assumptions leading to architectural mismatch is *precisely* $\mathcal{NL}$. Put differently, the question is whether, for a given non-local design statement φ, *any* implementation that satisfies φ can be expanded to violate φ.

Finally, it is of interest to explore a syntactic reformulation of the Locality criterion within the confines of a specific specification language, as delineated in our discussion in Section 2.4.

([3]) In [6] we refine $\mathcal{L}$ using the Intension criterion.

References

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

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

[3] J.O. Coplien. "Idioms and Patterns as Architectural Literature." *IEEE Software*, Vol. 14, No. 1 (Jan. 1997), pp. 36—42.

[4] J. Derrick, E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Berlin: Springer-Verlag, 2001.

[5] A.H. Eden, Y. Hirshfeld. "Principles in Formal Specification of Object Oriented Architectures." *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research – CASCON*, Nov. 5—8, 2001, Toronto, Canada.

[6] A.H. Eden, R. Kazman, Y. Hirshfeld. "Abstraction Strata in Software Design." Technical report CSM-411, Department of Computer Science, University of Essex, June 2004.

[7] A.H. Eden, R. Kazman. "Architecture, Design, Implementation." *Proceedings of the 23rd International Conference on Software Engineering — ICSE*, May 3—10, 2003, Portland, OR.

[8] M. Fayad, D.C. Schmidt. "Object-Oriented Application Frameworks". Guest editorial, *Communications of the ACM*, Vol. 40, No. 10 (Oct. 1997).

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 2003.

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

[11] D. Garlan, R. Allen, J. Ockerbloom. "Architectural Mismatch or Why It's Hard to Build Software From Existing Parts." *Proceedings of the Seventeenth International Conference on Software Engineering – ICSE'95*, Seattle, WA, Apr. 1995.

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

[13] D. Hou, H. J. Hoover. "Towards Specifying Constraints for Object-Oriented frameworks." *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research – CASCON*, Nov. 5—8, 2001, Toronto, Canada.

[14] R. Johnson, B. Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming*, Vol. 1, No. 5, Jun./Jul. 1988, pp. 22—35.

[15] R. Johnson, B. Woolf. "Typed Object". In: R. Martin, D. Riehle, F. Buschmann (eds.) *Pattern Languages of Program Design 3*. Reading, MA: Addison-Wesley, 1997.

[16] 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*, Denver, CO, Nov. 1—5, 1999.

[17] V. Matena, M. Hapner. *Enterprise JavaBeans™ Specification, v1.1*, 1999. Palo Alto, CA: Sun Microsystems.

[18] R.T. Monroe, A. Kompanek, R. Melton, D. Garlan. "Architectural Styles, Design Patterns, and Objects." *IEEE Software*, Vol. 14, No. 1 (Jan. 1997), pp. 43—52.

[19] D.E. Perry, A.L. Wolf. "Foundation for the Study of Software Architecture." *ACM SIGSOFT Software Engineering Notes* Vol. 17, No. 4 (Oct. 1992), pp. 40—52.

[20] W. Pree. *Design Patterns for Object-Oriented Software Development.* New York, NY: ACM Press, 1995.

[21] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley & Sons, 2000.

[22] M. Shaw. "Some Patterns for Software Architectures." In J. Vlissides, J.C. Coplien, N.L. Kerth (eds.) *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[23] C. Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. Reading, MA: Addison-Wesley Professional.

[24] R. Turner. "The Foundations of Specification I". Technical report CSM-396, Department of Computer Science, University of Essex, 2004.