10-2006

# A Pattern Language for Extensible Program Representation

Andrew P. Black
*Portland State University*, black@cs.pdx.edu

Daniel Vainsencher
*Portland State University*



## Citation Details

Vainsencher, Daniel, and Andrew P. Black. "A pattern language for extensible program representation." Proceedings of the 2006 conference on Pattern languages of programs. ACM, 2006.

# A Pattern Language for Extensible Program Representation*

Daniel Vainsencher[1] and Andrew P. Black[2]

[1] The Technion
danielv@techunix.technion.ac.il
[2] Portland State University
black@cs.pdx.edu

**Abstract.** For the last 15 years, implementors of multiple view programming environments have sought a single code model that would form a suitable basis for all of the program analyses and tools that might be applied to the code. They have been unsuccessful. The consequences are a tendency to build monolithic, single-purpose tools, each of which implements its own specialized analyses and optimized representation. This restricts the availability of the analyses, and also limits the reusability of the representation by other tools. Unintegrated tools also produce inconsistent views, which reduce the value of multiple views.

This article describes a set of architectural patterns that allow a single, minimal representation of program code to be extended as required to support new tools and program analyses, while still maintaining a simple and uniform interface to program properties. The patterns address efficiency, correctness and the integration of multiple analyses and tools in a modular fashion.

## 1 Introduction

Our view of programs is that they are not linear text but complex, multi-dimensional structures [5]. Our view of programming environments is that they are tools to reveal and elucidate this structure. As programs become more complex, it is thus not surprising that integrated program development environments (IDEs) are growing in importance and sophistication.

This view of programs and environments is not new, but in recent years it has become more widely accepted. The agile development community, for example, emphasizes the importance of incremental design, which implies frequent refactoring of the code base. Refactoring tools require deep knowledge of the structure of the program, and to be maximally effective they also require ways of displaying that structure to their users [32]. In short, the agile revolution could not have happened without supportive tools.

* A preliminary version of this article was workshopped at Pattern Languages of Programming (PLoP) 21st–23rd October 2006, Portland, OR, USA.

### 1.1 IDEs are Ecosystems

Integrated development environments (IDEs) are ecosystems in which three different "species" — communities of contributors — meet and interact. Two of these species are easy to identify: the *architects* of the development environment itself, and the *toolsmiths*, who build plugins and components to support some user task. The third species comprises the program *analysts*: the people who write the code that builds the parse trees, ASTs, dependency graphs, and other structures that the toolsmiths need to do their job.

The biological metaphor is imperfect because the same contributor may belong to more than one of these species, but it useful to distinguish them, because the contributions require different expertise. Nevertheless, powerful forces urge these three species to cooperate. Their works are synergistic: a widely used IDE provides the analysts with a platform; the availability of the analyses makes it simpler for the toolsmiths to build more sophisticated tools; the availability of more tools makes the IDE more attractive to programmers, and having more programmers use the IDE is the goal of the architects. All three species therefore collaborate to design the interfaces through which the tools, analyses and the architectural framework interact.

Getting these interfaces "right" — or at least right enough — is not easy. We believe that merely recognizing that there are three inter-dependent communities helps in the process, because the vital role of the analyst has not always been recognized. Traditionally, some basic analyses were built into the architectural framework; if a particular tool needed a deeper analysis, or just a different one, the toolsmith would incorporate this analysis into the tool itself. In effect, the analyst was excluded from the ecosystem.

We became involved in this work because we were building *Multiview* programming environments — IDEs in which *multiple* tools provide different views on the *same* program elements. This was easy to do when the architect had provided a shared code model that exposed all the information needed by the views. However, as our views started to expose latent properties of the program — properties whose calculation required significant analysis — we were faced with the problem of where to put the analysis code. Putting the analysis into the tool would not work: the results of the analysis had to be shared by several tools. It seemed clear that the right place for the analysis was in the code model itself: all we had to do was invent a "universal" shared code model that was general enough to provide the data for all views. Such a shared model would need to make available not only all of the information present in the code directly, but also all of the information that could be inferred from it, just in case some tool might need it. This may be difficult, but has the important advantage of ensuring that all of the views are consistent.

This article advocates such a shared code model. We are by no means the first to do so; as long ago as 1991 Scott Meyers wrote: "many problems ... would be solved if all the tools in a development environment shared a single representation ... Unfortunately, no representation has yet been devised that is suitable for all possible tools" [28].

In three years of work, we also failed to devise a general, abstract and efficient shared code model suitable for all possible tools. With the benefit of hindsight, we believe that the task is impossible: generality and efficiency are almost always at odds with each other. It now seems obvious that what we need instead is an *extensible* code model, so that generality can be added when it is found to be needed, but not before. Moreover, by focussing the computationally expensive analyses on those parts of the code base that the programmer actually finds interesting, we can avoid wasting cycles computing information that will never be used.

Instead of searching for a "silver bullet" code model for each programming language — a model that will satisfy all of the possible needs of toolsmiths, analysts and architects — we need an architecture and a common language that allows them to build on each others' work in mutual co-dependence. It is the beginnings of such an architecture and language that we seek to provide in this article.

### 1.2 The Pattern Language

The main contribution of this article is a pattern language for an abstract, extensible and efficient shared code model. The patterns are presented in four groups. The first group, described in Section 3.1, answers the primary question posed above: how should the responsibilities of a multi-view programming environment be divided among the code model and the tools that maintain the views? The second group (Section 3.2) presents some common categories of information that are strong candidates for integration into the code model. The third group is devoted to performance considerations, and how these affect the proposed design (Section 3.3). One unfortunate consequence of performance work is that it may add difficult-to-find bugs; the fourth group of patterns (Section 3.4) is about removing them.

**List of patterns**



The essence of the pattern language is to apply the model-view architecture to program development environments. The code — *and all of the interesting analyses on it* — becomes the model; the various tools in the environment become views that do nothing more than ask the model for the data that they need and present it on the screen, using *Observer* to ensure that they are notified of changes. This architecture hides the distinction between those attributes of

the code that are stored explicitly in the model (or can be computed from it immediately), and those attributes that must be derived from a code analysis, perhaps at considerable computational cost. The key to accommodating all of these analyses in the model — including the ones that we haven't yet realized are interesting — is to make the model extensible. Some of these analyses will be complex, and will expose global properties of the code. And yet: each model extension must be able to answer, at any moment, any sequence of questions about the code that a tool might ask. Moreover, it must do so quickly, so that the view can respond in real-time as the code is modified.

The benefits of this architecture to the toolsmith should be obvious: the toolsmith no longer has to know which attributes are part of the model and which are calculated by analyses, and if multiple tools need to use similar analyses, the results are easily shared.

In the Eclipse code model, in contrast, clients must be aware of this distinction between stored and computed attributes. For example, the interface org.eclipse.jdt.core.IType, representing a Java Class or Interface, has methods getSuperclassName() and getSuperInterfaceNames(), but it does not have methods to return the type's subclasses or subinterfaces. The toolsmith who needs that information must build an ITypeHierarchy object and invoke its getAllSubtypes(IType) method. The ITypeHierarchy is not automatically updated when the model is changed; that too is the client's responsibility. The toolsmith is also warned that "once a type hierarchy has been created, it is more efficient to query the hierarchy for superinterfaces than to query a type recursively. Querying an element performs a dynamic resolution, whereas the hierarchy returns a pre-computed result"[12]. Thus, for optimal efficiency, the client needs to keep track of whether an appropriate ITypeHierarchy object has been computed (in which case it should be used), or not (in which case the IType must be queried directly).

We hypothesize that the reason that the Eclipse JDT expects its clients to be aware of all of these details is that hiding them was deemed to be too expensive. Of course it would have been possible to build all of the functionality of an ITypeHierarchy into the base model, and to refresh it every time that the model changed. But the computational cost would be high, particularly as programs become large. Moreover, much of the information thus computed would never be used: making programmers using the IDE wait at the keyboard for information that they do not want to see is a bad idea.

For this reason, our patterns pay particular attention to performance. Without a way to reduce the cost of computing derived information, the architecture proposed here would be nothing more than a hollow shell, attractive in the abstract, but completely infeasible in practice.

In the next section (Section 2) we introduce a particular code model extension, which we have implemented in the Squeak environment for Smalltalk. We use this example throughout the article to illustrate the issues and how the patterns address them. But first, we offer an apology.

### 1.3 Patterns or Proto-patterns?

In the introduction to Linda Rising's collection *Design Patterns in Communications Software* [36], Douglas Schmidt writes:

> Patterns represent successful solutions to challenges that arise when building software in particular contexts. When related patterns are woven together, they form a pattern language that helps to (1) define a vocabulary for talking about software development and integration challenges and (2) provide a process for the orderly resolution of these challenges. [36, p. xii].

While there is no one definition of the term "design pattern", a useful rule of thumb, as Schmidt indicates, is that a pattern presents a solution to a problem in a context. Another criterion is that a pattern should not seem startlingly new to practitioners: on the contrary, the expected response to a pattern is: "how elegant; I might have thought of that myself, if I had been faced with that problem" or "right; I have done that before in other contexts, and I see that it might be useful here too". The purpose of presenting design ideas in pattern form is to define a language for architecture in a specified domain, and to open a dialog in and around it. The language that we present here does not pretend to deal with all possible problems in the space of program representation, but we believe that it is self-contained, in that it does provide the terminology needed to talk about its own pattern, their consequences, and the concomitant solutions.

By all of these criteria, the pattern form is appropriate for this work. However, there is commonly an expectation that a pattern distills from multiple experiences. For example, Buschmann *et al.* [7] propose finding at least 3 examples of an idea when pattern mining, and Gamma *et al.* [17] offer at least two examples of each pattern. By this criterion, some of the strategies that we propose do not yet qualify as patterns because we cannot offer evidence that they are currently in wide use. However, we feel that these strategies are more than proto-patterns: we have applied them to a difficult programming problem, and found that things suddenly became easier. Our implementation was in the context of the Smalltalk programming environment, but the patterns are certainly not language-specific. We have found some (but not all) of them adopted in other environments, for example, the Eclipse environment for Java. We feel that reporting on our experience and presenting these patterns now will enable more development environments to build on this architecture in the future, and in the process extend and evolve our contribution into a pattern language that covers an ever-increasing part of the problem space. In this spirit, we particularly welcome additional examples for, or counterexamples to, our putative patterns.

Having raised this issue, for conciseness we will nevertheless refer to a specific proposed solution as a pattern in the remainder of this article.

## 2 A Running Example

Our implementation of these patterns has so far taken place in Squeak Smalltalk, where we have been working on tools to support traits [6, 40]. Although Smalltalk

has no explicit syntactic marker that identifies an abstract class, abstract classes are widely used in practice. They can be identified because they are missing critical methods. An example is the class Collection, which is the abstract superclass of many of the concrete kinds of collection, such as Sets, Bags and Dictionaries. Collection is abstract because it does not provide implementations for add:, remove:ifAbsent:, and do:; it is the responsibility of its subclasses to provide these methods. This is indicated by the existence of explicit *marker methods* on these messages, *i.e.,* methods with the body self subclassResponsibility. (Such a method will raise an error if it is ever invoked.) Collection *does* provide concrete methods for addAll:, remove:, collect:, select:, *etc.*, which are all ultimately implemented in terms of the abstract methods.

However, not all abstract methods are indicated by marker methods like subclassResponsibility. An examination of the methods provided by class Collection also reveals that a message with selector atRandom: is sent to self in another method, even though there is no maker method indicating that atRandom: is abstract: atRandom: is an *implicit* abstract method. Thus we see that an analysis of the whole of the class Collection can reveal that the class is abstract, and can also infer the names of the four abstract methods that must be provided by subclasses. However, this analysis can be computationally intensive for a large class or a deep inheritance hierarchy.

While programming in Smalltalk, we have found it very useful to show, in real time, which *classes* are abstract. When viewing a particular class, it is also useful to show a list of the abstract *methods* — which we call the *required methods* of the class, or simply its *requirements* [39]. Inferring the required methods supports "programming by intention," a style of programming in which methods are used before they are written [24]. The constant display of the required methods acts as a "to-do list" for the programmer, reminding her of the methods that she has yet to write. In Figure 1 the Smalltalk browser is showing the required methods of Collection. We call this the "requires view"; it is an example of a view that reflects the result of an extensive non-local analysis of the code base.

In seeking to implement the requires view, we started out with Schärli's efficient algorithm for determining if a particular method is required by a specific class and its subclasses [37, 39]. Our problem, then, was to construct from this algorithm a practical browser that would indicate which classes in a possibly long list of classes were abstract, and which methods were required by a particular class.

It turns out that when programmers use browsers they frequently change the display of the list of classes. The naïve approach of running Schärli's algorithm on every defined selector on every class in a long list was far too slow: the results were not available within the 100 ms time box that is the norm for interactive response. Our problem was how to efficiently reify the information needed for the *requirements* calculation in the code model so that this information could be shared amongst various tools, without repeatedly recalculating it.

By "reify the information" we mean "make the information concrete". In a sense, the implicit information is there in the code model all the time, but

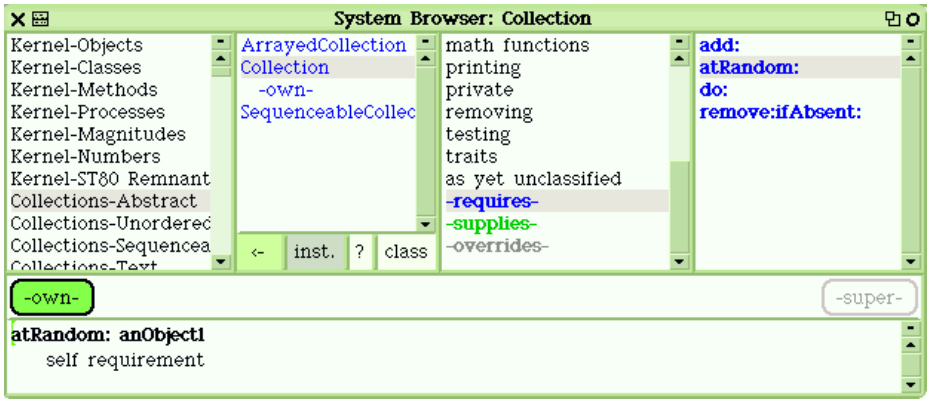


Fig. 1: The Smalltalk browser showing abstract classes and required methods. In the method list pane (at the top, on the far right), all four of the required methods of class Collection are listed. In the class pane (at the top, second pane from the left), abstract classes are highlighted in blue. The list of required methods and the fact that a class is abstract are both deduced by code analysis.

a lot of computation is required to extract it. Reified information, in contrast, is directly available through an appropriate method with little computational overhead. An additional problem was that Schärli's algorithm itself required walking the inheritance hierarchy, and obtained part of its efficiency from the careful use of caches to avoid recalculating on behalf of a subclass the same results that had already been calculated for its superclass. We hoped to be able to reuse these caches in a more general setting, so that the cached information would become available to other tools as part of the model, rather than being the exclusive property of one algorithm.

## 3 The Patterns

As we mentioned in Section 1.2, we have arranged our description of the patterns into four groups. The first group (Section 3.1) describes the division of responsibilities between the code model and the tools that use it; the second group (Section 3.2) addresses the content of the extended model; the third group (Section 3.3) provides guidance on making the extensible model fast enough, and the fourth and final group (Section 3.4) addresses correctness. Figure 2 shows the relationships between the patterns and the problems that they address. When we use the names of these patterns in the body of this article, we will set them in a *slanted sans-serif font*.

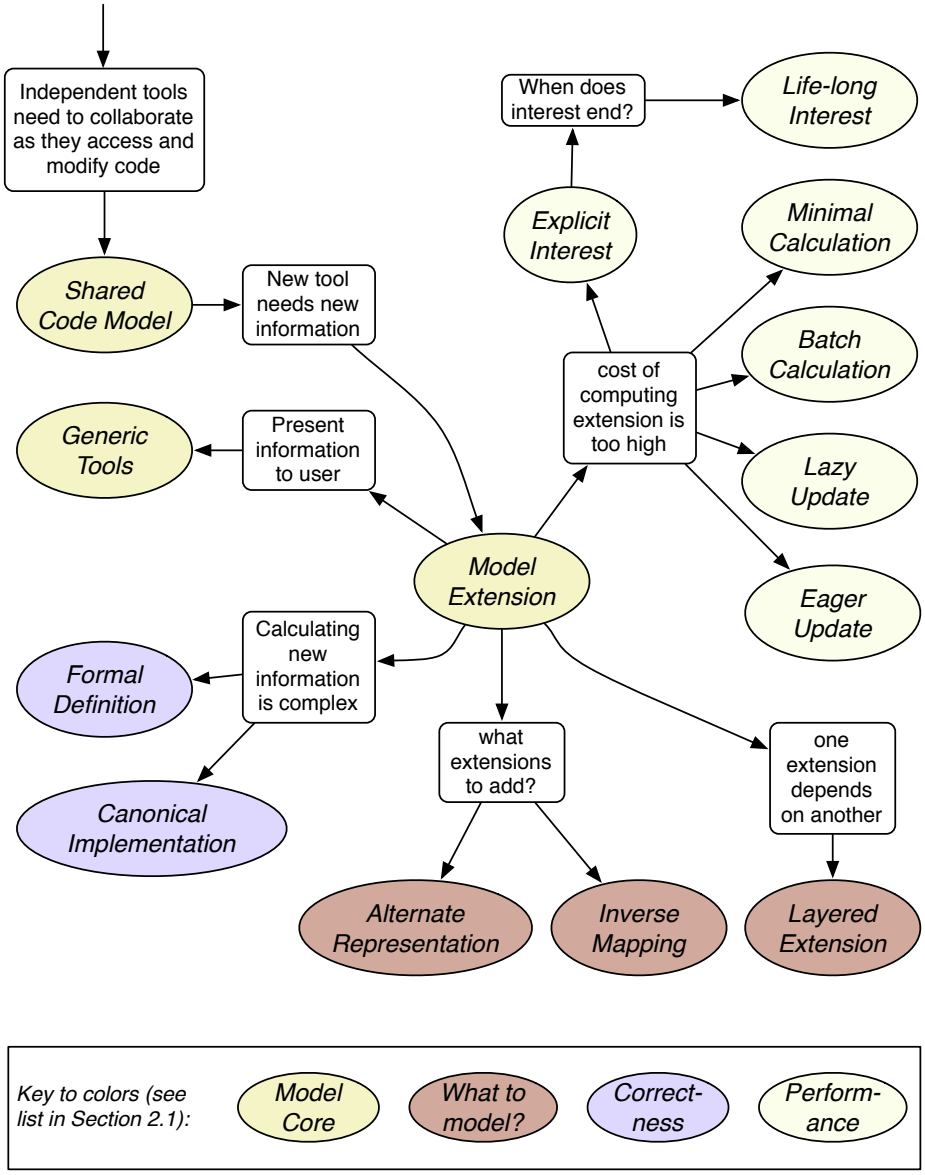


Fig. 2: Text in rounded rectangles summarizes a problem. Shaded ellipses name patterns discussed in the body of the article; they are colored by group. Arrows show how problems are addressed by the patterns, and how the use of the patterns gives rise to new problems.

### 3.1 A Code Model supporting Multiple Tools

Underlying any development environment is a representation of the code of the program under development. A very common scheme for this representation is code files in directories, possibly with additional files for metadata about the project. In contrast, Squeak Smalltalk [23], on which we implemented the requirements browser, uses a rather different code model, although it is one that is commonly used for Smalltalk. Our first pattern describes the structure of this code model and some of the features that contribute to its extensibility. The existence of these features in Smalltalk systems is not coincidental: there has been a long tradition in the Smalltalk community of programmers augmenting the set of available programming tools. *Shared Code Model*, described below, provides the foundation for programming tools that display information that is available explicitly in the model.

---

## Shared Code Model

---

**Context.** You are designing, refactoring or extending a program development environment that contains several independent tools giving different views on the same code.

**Problem.** How can you ensure that each tool is up-to-date regarding all of the changes made in every other tool, and that the tools are consistent with one another in their interpretation of the code?

**Forces.** The following forces are in play.

- The tools may be written by different developers.
- Multiple representations of the code increase maintenance costs, and lead to inconsistencies between the tools.
- The model needs to be *complete* enough so that all tools can be based on it.
- The whole development environment needs to be responsive to common actions, so requests on the model should be sufficiently cheap.
- The size of the screen and the capacity of the user's brain are limited, so the user will most often be examining and changing only a small number of program elements at any one time.
- Many tools may need access to the coarse structure of the whole program, for example, the type hierarchy.
- The tools may live in separate address spaces, in which case communication between the tools and the model will be costly.
- When one tool changes the model, other tools viewing the model need to be able to reflect the change promptly.

**Solution.** Maintain a single direct representation of the high-level syntax of the program as a graph of objects, organized to permit efficient browsing and searching. Lower levels of syntax need not be built as object structures; instead they can be kept as text.

Keep the shared code model *minimal* to avoid redundancy, and the complexity and inconsistencies that result from it. This means that information that can be calculated from the model quickly should not be cached in the model. Some other reasons for minimality are that specialized information is likely to be useless to most of the tools, and that a minimal model is simple to understand.

In order to keep the tools and other clients up to date, the code model must implement a notification mechanism, such as an *Observer* [18]. The notification events should include enough information for a tool to update the relevant parts of its output efficiently. In Squeak this information includes the identity of the code model element (*e.g.,* the method or class) that changed, and the nature of the change (addition, removal, or modification).

**Consequences.** This solution has the following consequences.

- The *shared* nature of the code model and the use of *Observer* to notify clients of changes together allow multiple tools to remain synchronized and yet oblivious of each other.
- The representation of the high levels of code as objects makes common navigation tasks easy. For example, it is easy to access the superclass of a class, or to enumerate the methods of a class.
- Detailed, structured representations of small parts of the program can either be stored as part of the *Shared Code Model*, or can be computed on demand from a textual representation.
- Repeatedly computing an alternative representation on demand may be too slow. Rather than forcing clients to cache this information themselves, cache it in the shared code model as an *Alternative Representation* (p. 20).
- The *Shared Code Model* may turn out not to be complete: new tools may require information about the program that you did not anticipate. In this case, implement a *Model Extension* (p. 13).

**Implementation.** One of the decisions that must be made when applying *Shared Code Model* is which parts of the code to represent as structured objects and which parts as linear text. A second decision is what to put in main memory, and what to leave on the disk. Along with the choice of representation, this will obviously dictate the memory footprint of the model, and thus the scalability of the environment.

The two decisions are not entirely independent, because the operation of following an object reference on the disk is roughly 10 000 times slower than following a reference in main memory, and so disk storage is much more suitable

for sequential representations, and main memory for linked ones. In Squeak, the class hierarchy and compiled methods are stored as an object graph in main memory, whereas method bodies are represented by pointers to text on the disk. An environment for the manipulation of very large programs might be forced to keep more information on disk; in this case various kinds of database index structure could be used to improve access times.

**Known Uses.** Possibly the first development environment to use a structured code model was the Cornell Program Synthesizer [41], which represented programs as threaded "executable derivation trees". This model was "shared" by the editor and the interpreter. The Synthesizer was targeted at beginning PL/I programmers; it seems unlikely that it could handle commercial-size programs.

The Smalltalk-80 development environment [21] implemented a *Shared Code Model*, and this model is alive and well in its decedents, such as Squeak. However, it is not as complete and straightforward as might be expected, principally due to the limited memory on the early Smalltalk-80 systems. For example, the navigation from a class to its methods and back again is rather convoluted, reflecting some implementation decisions that ought to have been better hidden.

The Cadillac programming environment [14], developed at Lucid for C++ and later named Energize, also had as its goals easy extension by the builders of the environment, and tight integration of the tools as perceived by the user. It achieved these goals by defining tool protocol interfaces that could be used to access a shared code model of persistent objects that were stored either in ISAM files or in an object-oriented data base.

The most well-known contemporary example of a *Shared Code Model* is in the Eclipse Java Development Toolkit (JDT), which is designed to be extensible [16, 34]. The JDT includes a model of Java code in which objects (implementing the interface IJavaElement and its subtypes) represent elements of the program under development. It provides a way to examine the code, and includes a change notification mechanism. The Java model is computed from the source code on demand, and then cached in subinstances of class JavaElementInfo. One consequence is that programmers must be aware of this caching; Gamma and Beck warn that "finding a method by traversing the Java model starting at the root and then drilling down to methods is expensive. It will fully populate the element info caches." [16, p. 315]. However, when used as intended, at least one commercial IDE developer reports that performance is satisfactory even for queries that require traversing the code of a whole project.[3]

The Eclipse JDT Java model thus fits many aspects of the *Shared Code Model* pattern. However, we cannot say that the Eclipse JDT follows the intent of this pattern, because it also contains at least two other code models: the AST model (ASTNode and its subtypes), which represents the syntax of a single compilation unit, and the bindings model (IBinding and its subtypes), which represents the

[3] Brian Wilkerson, Instantiations, Inc. Personal communication.

network of connections reachable from a compilation unit. The intent of *Shared Code Model* is that there be a *single* model on which all tools can rely.

According to some of our reviewers, the C++ language component of Eclipse called CDT also does not have a single shared code model. It instead uses several separate representations of the code, making it difficult to decide which to use for particular purposes.

**Related patterns and variants.** Riehle *et al.* [35] present the Tools and Materials metaphor, which motivates the distinction between the application (a tool) and the model on which it operates (the materials), and defines particular interfaces between them. In terms of this metaphor, our pattern language aims to make the code model a better material, by moving the boundary between it and the tools. While both approaches are intended to make the environment more extensible, they address different aspects of the problem, and make different assumptions. For example, we strive for the ability to add properties to a fixed set of code elements, while they consider the set of properties fixed and make the system extensible with new kinds of materials that enjoy those properties.

Other patterns such as MVC [7] also make the distinction between the model and the tools that allow the user to operate on it.

We have already mentioned the role of the *Observer* pattern [17] in connecting the code model to the tools that operate on it, and the use of *Alternative Representation*, described on page 20, when it is necessary to make available two different representations of the same program element.

□

It was our goal to implement a tool that uses some information that the Squeak code model does *not* provide explicitly: the *requirements* of a class. We wanted to access the requirements in at least two places: first, to annotate a class as being abstract when its name appears in the browser's list of classes, and second, to display the requirements of a class when the programmer browses that class. In addition, because there were other kinds of code browser that might be extended to use the *requirements* property, we wanted to make it very easy to access this property — as easy as it is to access the base properties of the code model. For example, getting the names of the instance variables of the class Morph in Smalltalk is very simple, because instance variables are explicit in the base model: the programmer merely issues the query Morph instVarNames. Our goal was to provide access to the requirements of a class using an equally simple query: Morph requiredSelectors.

However, our starting point was quite different: Schärli's algorithm, for good performance reasons, was implemented to update a global cache of requirements. So, getting up-to-date values required first updating the cache for the class in question, and only then accessing it. So, if we were interested in the requirements of class Morph, the code that we had to write was

```
Morph updateRequiredSelectors.
requirements := Morph requiredSelectors.
```

Thus, clients had the responsibility of ensuring that the cache was up to date: this was both inconvenient and error-prone. We felt that this was the wrong trade-off, and that simplicity of interface was more important than simplicity of implementation [13]. The next pattern, *Model Extension*, shows how to retain both.

## Model Extension

**Context.** A development environment uses a *Shared Code Model* to represent the code and includes several tools that operate on it.

**Problem.** How do tools access properties that are not stored in the code model, but are calculated from it? How can such properties be shared by multiple tools?

**Forces.** The following forces are in play.

- Many of the tools in an IDE exist to access properties and structures that are *implicit* in the code, and therefore not present in a minimal shared code model. Nevertheless, users of the code model would like to be able to interrogate these properties and structures through a simple and concise interface.
- Despite the fact that several tools may wish to access the same properties and structure, the same calculations should not be repeated.
- Analysis algorithms should not be implemented multiple times; the same implementation should be available to multiple tools.
- One of the ways in which a new analysis can add value is by defining new properties or making visible a new level of structure, so these parts of the code model should be open to extension.
- The implementation of an analysis may be complex; it should be encapsulated and separated from the core of the model.

**Solution.** Express each new property as an extension to the interface of an appropriate class in the *Shared Code Model*. Implement the calculation of this property as a *Model Extension*, that is, place it outside the *Shared Code Model* itself, but place an interface to it inside the code model.

In our running example, the interface that we desire is that the class model understand the additional message requiredMethods. This will make the set of

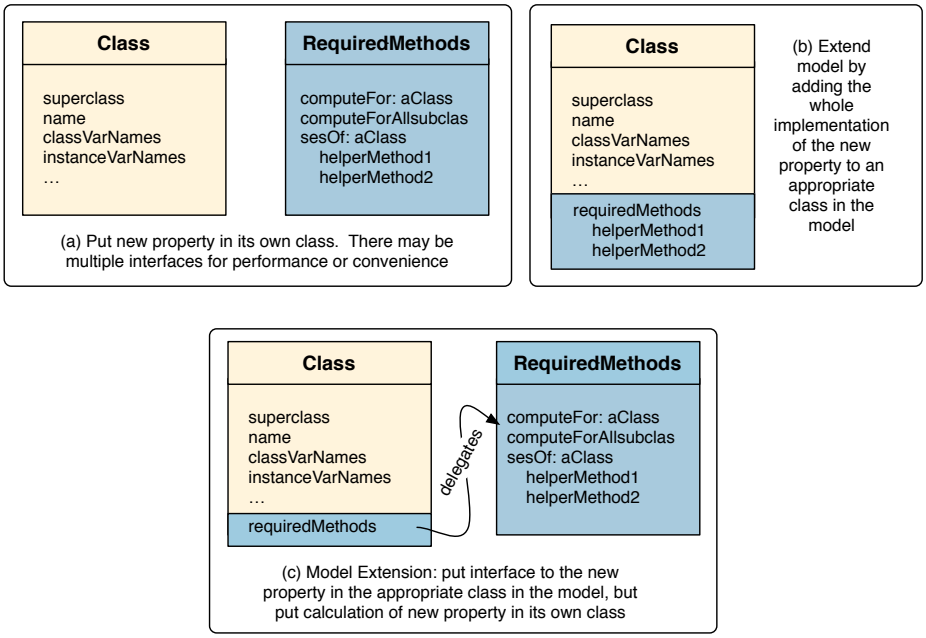


Fig. 3: Here we show three alternatives for enriching a *Shared Code Model*. In (a), the algorithm to extend the model (shaded buff) been added in a separate class (shaded blue). In (b), the whole of the implementation of the extension has been placed inside the model class. Alternative (c) exploits *Model Extension*: the *interface* to the extension is in the appropriate model class, but the *implementation* is encapsulated in its own class. See *Model Extension* (p. 13) for more details.

required methods accessible in the same simple and direct way as other properties of the code model: by sending a message to the appropriate object in the model.

Figure 3 shows diagrammatically three alternative organizations for the code necessary to support this new property of Class. In Fig. 3(a), Class is part of the shared code model, shaded buff. The algorithm to compute the required methods has been added in a separate class, shaded blue. This means that the user of the extension must be aware that it *is* an extension, and must learn a special-purpose interface for accessing the information that it represents. The user may also need to be concerned with initializing and finalizing the classes that implement it.

In (b), the whole of the implementation of the extension has been placed in the existing model class. This makes it possible to present a uniform interface, but fails to encapsulate the new algorithm, and makes it hard to provide (potentially more efficient) alternative interfaces. It also requires that the implementation language support a very complete form of class extension.

Alternative (c) illustrates *Model Extension* (p. 13): the *interface* to the extension is in the appropriate model class, but the *implementation* is encapsulated in its own class. Only the most modest class extension mechanism is required.

To the tool writer, extensions like requiredMethods add richness to the otherwise minimal *Shared Code Model*, making it a more useful representation of the code. By accessing *all* aspects of the code through the code model, the implementations of the various tools become simpler, and the interfaces to the output of the tools become more uniform. This makes it easier to use, understand and maintain the various tools.

Although Figure 3(c) shows the implementation of the extension as a single class, more complex properties might require many classes. The intent of the pattern is to separate the implementation of the extension from the interface in whatever way is most appropriate, allowing authors of model extensions to address the complexities of correct and efficient computation of each property without increasing the complexity of the *Shared Code Model* itself.

The extensions to the interface of the code model demanded by this pattern should be *class extension* methods if the implementation language supports open classes [8, 9], or Extension Objects [15] it it does not.

**Consequences.** This pattern says that the derivation of properties by interpretation of the *Shared Code Model* should be an extension to the code model itself rather than a part of the tool that needs to use the properties. The consequence is that these derived properties are made available to other tools. This synergy between tools enables a rich suite of tools to be developed more quickly and with less effort. At the same time, the *Shared Code Model* itself remains simple, and the interfaces to direct and derived properties can be consistent.

**Implementation.** Many programming languages do not recognize the importance of *open classes* [8]; they make it hard to *extend* classes that have already been written by another programmer or obtained from an outside organization. In Java and C++, for example, any extension to the interface of a standard class library requires *editing* (rather than augmenting) its implementation; this may be impossible unless the implementation is under the control of the extender. Some Java variants such as MultiJava [8] and AspectJ [25] recognize the value of open classes and do provide for class extension; programmers using ordinary Java can fall back on the *Extension Object* pattern [15].

Smalltalk supports open classes: the Smalltalk programmer can extend the interface of the code model by adding new methods in a separate package that contains what is known as a class extension. However, Squeak's class extension mechanism does not allow an extension package to modify the *representation* of an existing class, that is, an extension package can add a method, but cannot add an instance variable.

This pattern places all of the logic of the *Model Extension* in new classes. Thus, the only extension facility that it requires is the ability to add a new stateless method to an existing class. This mitigates the effect of any deficiency of the chosen implementation language.

**Known Uses.** In addition to our use of *Model Extension* to support the *required method* property, this pattern is also used in the Eclipse JDT to add higher-level functionality to primitive objects. However, because Java does not allow class extensions, an implementation of this pattern in Java is somewhat inconvenient.

To see how this pattern is used in Eclipse, consider the interface org.eclipse-.core.resources.IFile, which represents raw data and support methods like getContents() that make it possible to read that data. If the data represents a Java class, some tools may prefer to deal with a structured representation in which one can examine individual fields, the set of defined methods, and so on. This more structured interface is called IType.

Eclipse uses the Extension Object pattern to effectively add IType's methods to IFile. IFile implements the IAdaptable interface, which means that it provides the method getAdaptor(Class). Asking an IFile for an adaptor to IType answers an object that represents the same file, but which has type IType, and which will be able to respond to questions about the structure of the code in the file.

In addition to the non-uniform interface and the extra lines of code, Extension Object has a further disadvantage: its use must be anticipated. There is no way to proceed if the need arises to extend a class that does not implement IAdaptable, and thus does not support the getAdaptor(Class) method. For example, while IType implements IAdaptable, ITypeHierarchy does not.

**Related patterns and variants.** This pattern specifies the mechanism of interface extension, and the separation of interface from implementation, but says nothing about the form of the implementation itself. However, implementation is critical: although the architecture that we have specified so far is functionally complete, it is insufficient to allow implementations to obtain reasonable performance on code models of any significant size. Later we will describe the performance issues, and some patterns that help resolve them. The most important of these is *Explicit Interest*, described on page 28, which describes a simple interface addition that allows various implementation strategies, including the well known *Caching* [26].

□

Using the patterns that we have described, we implemented a *requirements browser* for Squeak, which flags classes that are abstract and displays their requirements. In the process we also implemented a reusable *Model Extension*, which is thus available to existing Smalltalk browsers — if they are extended to use it. However, implementing a simple interface that answers a useful question can be more rewarding than this: it should be possible to make use of a model

extension even without requiring that a tool developer write new tools or modify old ones. From the perspective of the user, discovering the existence and usefulness of a code analysis of which he had not previously been aware should not require him to learn to use a new tool. We can lower the costs of using model extensions to both tool developers and users by providing *Generic Tools* that are easily adapted to make use of new model extensions.

What do we mean by a "generic tool" and how could one have helped us with the *requirements* analysis? To answer this question, consider a generalization of the requirements browser. Our requirements browser displays in blue the names of classes that respond positively to the message isAbstract. It could easily be modified to use some other predicate to decide which classes to color. With a little more work, the browser could be made to show users a list of all of the predicates on classes, and allow them to decide *which* predicate will determine class coloring. Every new model extension that is a predicate would be added to this list automatically, making the extension immediately usable, even before a tool tailored to take advantage of the extension has been written. Even after tailored tools have been developed, any programmer opening the configuration dialog of the generic tool will see all the predicates defined on classes, and can choose to explore their usefulness without having to learn to use the more specialized tools.

Of course, the idea of *Generic Tools* is not limited to model extensions that are predicates. Ducasse and Lanza [10] describe a tool that allows for the *ad hoc* definition of code visualizations based on code metrics. In their tool, a code element is represented by a rectangle, and the values of different metrics determine its height, width and color. These metrics can be implemented as model extensions whose values are numeric.

Thus, we see that a new code analysis, if given a sufficiently abstract interface, provides us not just with one new tool or view, but with a new kind of information that can be used from many existing and familiar views and the tools that implement them. We capture this idea in the pattern *Generic Tools*.

---

## Generic Tools

---

**Context.** A development environment exists, various *Model Extension*s are implemented, and specialized tools use them.

**Problem.** How can you make the views of the program corresponding to these model extensions accessible without requiring tool-builders to write new tools, and application programers to learn to use them?

**Forces.** The following forces are in play.

- Many useful code analyses have been devised by experts on static analysis and published in the literature.
- These analyses are understood by few practitioners and used by even fewer — even though tools exist to apply them.
- The problem is that activating a specialized tool just to access an analysis takes time; this makes the analysis less valuable. Even more importantly, activating a specialized tool disrupts the programmer's workflow, forcing the programmer to divert her attention from the task at hand. Finally, the specialized tool will be unfamiliar, and the programmer must learn how to use it.
- For the developer of a code analysis, being forced to also create a tool that displays the results of that analysis raises the barrier to entry.
- While some code analyses result in specialized models of the code, many others result in representations of the code in terms of ordinary data types such as booleans, numbers, collections and graphs. Such data can be visualized and manipulated using general-purpose tools that are not dependent on the meaning of the data.

**Solution.** Instead of creating tools that are specialized to show a specific model extension to best effect, create generic tools that allow a programmer to make opportunistic use of a variety of model extensions. Such tools should provide a generally useful mechanism for using information about code. Generic tools should be extensible, that is, they should be able to use information from different sources.

Note that the requirement here is extensibility by a programmer, which can be achieved by different means at different levels. A tool framework might be extended to use a new *Model Extension* by coding a simple plugin. In some cases, a configurable tool might be extended without any programming, by merely specifying the name of an extension. The main requirement is that the writer of a *Model Extension* can make it visible from tools with small amounts of effort.

**Consequences.** A development environment with a suite of *Generic Tools* encourages the writing of reusable code analyses; exposing the outputs of code analyses also encourages the addition of new specialized tools that make use of them.

Note that *Generic Tools* are likely to sacrifice some aspects of usability. This is because they give the programmer a generic user interface rather than one specialized to a particular task. We might deduce that generic tools are more likely to complement than to replace more specialized tools. However, the user can be overwhelmed by a plethora of tools: a small number of more powerful, generic tools might actually be more friendly to the user, who is, after all, a practicing programmer.

**Known Uses.** We present two more examples of *Generic Tools*. First, we observe that a unit testing framework such as SUnit [2] can be thought of as a generic tool. SUnit allows convenient, reproducible checking of assertions about code. The assertions are usually about code functionality, and are checked by running application code on examples with known results. However, the assertions can also be about structural properties of the code, in which case they can be checked by sending messages to the objects in the *Shared Code Model*. Any new model extension can be used in this way immediately, without writing any new tools.

For example, unless one is building a framework extensible by inheritance, it is reasonable to expect that any abstract classes will have at least one concrete subclass. Leveraging the *requirements* model extensions and SUnit, this assertion can be expressed as follows.

```
(MyPackage allClasses) do: [ :each | self deny:
      (each subclasses isEmpty and: [each isAbstract]) ]
```

A suite of SUnit tests containing assertions like this may be thought of as turning the SUnit test browser into a general tool for maintaining invariants about the structure of the code. Naturally, such structural test suites complement, rather than replace, conventional test suites that check the function of the code.

The second example is the StarBrowser [44], which was designed by Wuyts *et al.* specifically to allow unanticipated integration of new properties. It displays a tree of nodes that can represent different types of objects, including code model elements. The StarBrowser can be made to display a specific set of child nodes, for example the required methods of a class, in an *ad hoc* fashion: one passes it an anonymous function that returns the set of nodes. It can also be customized more systematically, by creating a glue class that defines a new menu item on the appropriate node type. The latter kind of customization can be undertaken by a tool integrator who is the author of neither the StarBrowser nor the model extension. Thus, the working programmer can be exposed to the model extension through the StarBrowser without having to do the integration work.

□

### 3.2 Applications of Model Extensions

This section presents two general situations in which application of *Model Extension* (p. 13) can help to resolve a design problem; the patterns discuss the concerns specific to each, and how different *Model Extension*s may be composed. The discussion is heavy with performance concerns, because experience (both personal and vicarious) shows that performance can be quite critical in development environments that expose deep properties of the code in their views. In each pattern we therefore mention the performance problems typical to each kind of model extension, and point towards solutions. However, the reader might prefer to ignore this discussion on the first reading and focus on the effects of applying each pattern, rather than the details of its efficient implementation.

## Alternative Representation

**Context.** The development environment has a specific representation for each kind of element in the shared code model.

**Problem.** This representation is not the most appropriate for the specific tool that you are implementing. For example, the representation of a method might be textual; if you are implementing parenthesis matching, an syntax tree representation would be more convenient.

**Forces.** The following forces are in play.

- We cannot change the representation of the shared code model to suit our application.
- Because the *Shared Code Model* is complete, the information that we need is in the model somewhere.
- However, the information that we need is implicit; we would like it to be explicit.

**Solution.** Define the representation that you prefer as a *Model Extension* (p. 13). Calculate it when required in the conventional way, for example, build an abstract syntax tree by running a parser on the textual representation of the code. Efficiency may require the use of *Caching* [26].

**Consequences.** There is inevitably a cost in providing an alternative representation: time to compute it when needed, or space if it is to be cached, or possibly code complexity if some more efficient (but elaborate) solution is found. However, some of these costs are inevitable as soon as an IDE needs to expose aspects of the code that are implicit in the primary representation. For example, when implementing parenthesis matching, if we do not introduce the alternative syntax tree representation, we would instead need to do some form of ad-hoc parsing to find the parenthesis. This ad-hoc parsing has an execution cost, may need to be cached, and so on.

As with all *Model Extension*s, an *Alternative Representation* can be shared by multiple tools. Thus, this pattern helps to avoid code duplication, and reduces long-term maintenance costs. For example, the ad-hoc parsing code mentioned above would to some extent duplicate the normal parsing code that exists elsewhere.

**Known Uses.** The Squeak *Shared Code Model* represents methods as pointers into a text file kept on disk. This preserves all of the documentary structure of the source code [43], and provides access to the code that is fast enough for editing and browsing. However, an *Alternative Representation* is clearly needed for execution: Squeak uses bytecode. When the source code is edited (and when new code is read into the environment), it is immediately compiled to bytecode and the compiled methods are cached as objects. Other Smalltalk implementations, such as VisualWorks, compile directly to machine code. However, in all cases the existence of the cached of executable code is hidden from the user of the environment, who is given the consistent illusion of editing and manipulating source code.

*Alternative Representations* are commonly used by tools internally, even though they may not be exposed to other toolbuilders and users. For example, in addition to their target language, compilers commonly use a variety of intermediate representations, including control flow graphs, dependency graphs, and single static assignment form. These representations might be useful to programmers seeking to better understand their programs, but they are rarely exposed. Even if the compiler provides a way of exporting these representations, most IDEs do not make them readily visible. Compilers are only one source for potentially useful alternate representations; the program analysis literature has many more. Program slices, for example, can be very useful when debugging and transforming code [4].

□

We now move on to another application of *Model Extension*. We note that a common activity when reading code is navigating through chains of references; for example, in a procedural language, a procedure call is linked to the definition of a procedure by the scoping rules of the language. It is common for IDEs to automate this navigation, allowing the programmer to navigate from the call site to the definition with a single click. In an object-oriented language, an IDE may provide a pop-up menu on a message-send that allows the programmer to choose between the several correspondingly-named method bodies. This functionality is not particularly difficult to implement; it is enough to parse the particular method being browsed (this might be an *Alternative Representation*) and keep track of procedure and method definitions in the program.

Our next pattern is motivated by a related feature: the ability to navigate in the other direction, from a procedure to its call sites or from a method to the senders of the corresponding message. Navigating in this direction is useful when attempting to understand how a procedure or method is used, or when considering a refactoring such as adding a parameter. Note that answering the query in the naïve way (by searching for all calls to the procedure) would be expensive even if the parse tree of whole program were available, so an *Alternative Representation* is not the solution. Our proposed solution is captured in the next pattern, *Inverse Mapping*.

## Inverse Mapping

**Context.** The code model contains various natural reference structures. These are accessible via the *Shared Code Model* or by *Alternative Representations*, and used by programmers and tools.

**Problem.** How can we allow efficient navigation in the opposite direction?

**Forces.** The following forces are in play.

- Because the shared code model is minimal, many mappings that it supports directly will be one-way. For example, a class might have a direct reference to its superclass, but not to its subclasses.
- Both end-user tools and higher-level *Model Extensions* may wish to use these mappings in the other direction.
- Traversing the code base to search for such references is expensive.

**Solution.** Provide each *Inverse Mapping* as a *Model Extension*, so that the mapping is available to clients directly, but the implementation details are hidden.

Efficient implementation of inverse mappings can be complex. The basic difficulty is that finding all references to a code model element is a non-local task: references can be anywhere in the code base. This means that traversing the whole code base at each request is expensive. *Caching* the precomputed inverse mapping allows us to avoid multiple traversals of the whole code base for multiple queries. Then the basic difficulty shows up in a different way: to correctly answer queries about arbitrary reference targets this cached inverse mapping has to be of the whole code model, making it space-intensive for references that are common, such as procedure calls or message sends.

Using a cache also means that the cache must be kept up-to-date. Suppose that a procedure definition is modified, and corresponding changes are made to the appropriate part of the code model. To keep the cache current, we have to remove those entries corresponding to references that the procedure *used* to make, and add entries for the references that it *now* makes. The latter is straightforward, but the former poses a potential problem: how do we find the references that the previous version of a procedure used to make?

Sometimes this problem is solved by the change notification mechanism providing access to the *previous* definition of the changed element as well as to the new one. If this is not the case, it may be necessary to keep a separate record of all references in the forward direction, just for the purpose of cache invalida-

tion. In the worst case, it may be necessary to rebuild the cache from scratch by exhaustive search.

**Known Uses.** Squeak Smalltalk maintains the superclass of each class as an instance variable defined in class Behavior. It also caches the inverse mapping, the set of subclasses, in another instance variable (defined in class Class, a subclass of Behavior). When a class is redefined (possibly changing its superclass) the change notification mechanism provides the model with both the old and the new definitions, so the cache of subclasses can be updated incrementally.

However, Squeak is not always so organized: there are other places where this pattern could be applied, but is not. For example, it is easy to find the set of classes referenced by a class, but harder to find all the classes that reference a particular class. If this pattern were used in a Smalltalk environment, c referencedClasses would answer the set of classes referenced by c; The inverse mapping would be c referringClasses, which would answer all of the classes that refer to c. Whereas referencedClasses can be evaluated locally, by scanning the names referred to by all of c's methods, a naïve implementation of referringClasses would require a global traversal of all of the classes in the environment, searching for classes that refer to c.

At present, Squeak does not put *either* enquiry into the code model; instead the functionality to find all references to a class is implemented in the SystemNavigation tool (using global search). Another tool, the Refactoring Browser, implements the same functionality again. This illustrates the sort of problem that extensible code models will help to avoid.

**Consequences.** The maintenance of an *Inverse Mapping* might be expensive even with a sophisticated implementation. There is usually a tradeoff between space and time: making an inverse mapping available quickly will usually save time but cost space, because it will necessitate keeping a cache of an inverted index. However, notice that keeping a cache requires that the cache be re-validated when there is a change in any of the information on which it depends. If the re-validation process is very complex, it can also cost time. If maintaining an *Inverse Mapping* costs *both* space *and* time, use of this pattern may not be appropriate.

**Related patterns and variants.** As we mentioned above, the efficient implementation of an inverse mapping will sometimes require more than just a general cache. *Explicit Interest* makes it possible to maintain a more selective cache, for example, one that contains information only about references to specific code model elements. *Batch Calculation* and *Lazy Update* show us how to take maximal advantage of each non-local scan, particularly when a number of code model elements are updated at one time.

When offering examples of inverse mappings, we assumed a knowledge of the references from a method to other classes and messages. These references might be found in an *Alternative Representation*, created by analyzing the source code or by abstract interpretation of the bytecode.

□

We now return to our motivating example, the task of listing the methods required by a class. This means that we seek to implement a model extension *required methods*. A method $m$ can be required by a class $c$ either because $m$ is explicitly marked as required, or because $m$ is sent to self in a method of $c$ even though $m$ is not implemented by $c$ or its superclasses.

How can we find out whether a particular method is required in a class? The simplest thing that could possibly work would be the following. For every selector $s$ that may be required, scan every method implementation to see if it self-sends $s$. This defines a mapping *self sends* from methods to sets of selectors. This is a poor implementation technique: it would scan every method in every class many times. What we need here is the *Inverse Mapping* *self senders of* which provides, for each selector, the methods that self-send it. Thus, we find that we have quite naturally partitioned the complex computation of required methods into three layers of *Model Extension*, each built on a simpler one. This leads us to our next pattern: *Layered Extension*, described below.

## Layered Extension

**Context.** You have a definition of an interesting but complex *Model Extension*.

**Problem.** How do you implement this *Model Extension* efficiently, while at the same time promoting reusability?

**Forces.** The following forces are in play.

- Calculating this *Model Extension* requires as input other, expensive to compute information.
- This other information might itself be useful as a code property, available to other analyses.
- The calculation of the interesting and complex *Model Extension* can be broken down into a series of smaller, loosely coupled definitions.

**Solution.** Define each complex *Model Extension* on top of simpler ones, in layers. A higher-level property expresses *Explicit Interest* (p. 28) in the lower-level

extensions that it requires. Note that layering model extensions requires us to be careful in ordering recalculations. For example, we do not want to recalculate the *requirements* property before we have recalculated the *self senders of* mapping for the methods in the relevant classes. This ordering constraint can be addressed by *Lazy Update*, described on page 35.

**Implementation.** When implementing a *Layered Extension*, a number of competing forces come into play. For reusability, it is tempting to break the definition of an extension into small fragments. For performance, one needs to take care that each layer encapsulates a sufficiently expensive calculation to warrant the existence of that layer. For example, we found that in Squeak, caching the information "which superclass of $C$ implements a method on selector $s$" is useless, because the dictionary lookups required to access the cache were about as expensive as traversing the superclass chain to recalculate the information. While this might be regarded as commentary on a specific dictionary implementation, the larger lesson should be clear: application of these patterns complements, rather than replaces, good engineering and performance-oriented design.

**Related patterns and variants.** This pattern is a specialization of the *Layers* pattern [7]. When one *Model Extension* depends on another, it is important that they are recalculated in the right order. This is easy if the extensions are truly layered. If the dependencies become more complicated, one way to find an update order that respects the dependencies is to use *Lazy Update*, described on page 35.

□

In our running example, we created a *Model Extension* for the required methods of a class. We expect that as other code analyses are presented this way, more patterns will emerge, enriching this pattern language for discussing the implementation architecture of program analyses.

### 3.3 Making it fast enough

This subsection is devoted to performance, a topic we have mentioned several times, but haven't yet tackled seriously. A very important rule of thumb for optimization, codified in *Lazy Optimization* [1], is to avoid it until it proves necessary. This rule certainly applies here, and we advocate adhering to it.

*Why Performance Matters.* Why does performance play such a major role in our pattern language? It is our belief that efficiency considerations are an inherent part of this domain. Before we dive into detail, we will use an analogy to explain why performance is a pervasive problem, and how we alleviate it. The analogy is with the model-view architecture for graphical interfaces and the central role that the *Observer* pattern plays in making that architecture feasible.

The key idea in the model-view architecture is to decouple the model from the view. In an ideal world, the model will know nothing at all about the various

views: it will just get on with the business of representing its domain. Whenever a view needs to know some aspect of the state of the model, it asks. The problem with this naïve scheme is that the view needs to redraw itself on every display refresh cycle; it would be hopelessly inefficient to poll the model 60 or 80 times per second, usually to find that nothing has changed. The standard solution is to use the *Observer* [17] pattern, which requires the model to notify the views of any changes. The model is no longer oblivious to the existence of views, but it knows little or nothing about them beyond the fact that they exist. No description of model-view can be complete unless it shows how to address the performance challenge that arises from the decoupling of model and view.

The key insight to solving the performance problem of extensible program representations is that only a small part of the program is in view at any one time. Thus, the extensions need complete their analyses for that part alone. But how can the extensions know *which* part?

*Explicit Interest*, described on page 28, provides a way for a tool to notify the code model that *someone* is interested in a particular part of the code without the model having to care about *who* is interested.

*The Demands of Interaction.* When and how do issues of performance arise when computing a *Model Extension* (p. 13)? The most naïve implementation of an Extension would simply compute the property whenever it is requested. For properties that can be computed cheaply enough, this is a good choice: if computing the property is not much more expensive than retrieving it from a cache, there is no point in precomputing it. However, as properties start to be used in development environments to give interactive feedback to programmers, the meaning of "cheaply enough" is becoming more exacting. For example, whereas matching parenthesis used to be a by-product of the parsing step of a compiler, performed every few minutes, today most environments match parenthesis as they are typed. Similarly, whereas in the past it might have been thought sufficient to report missing "required" methods at release time, our desire to show this information interactively to support new styles of work [38] requires a much more sophisticated implementation. Thus, the frequency of use of an analysis can change from once every few minutes to multiple times per second.

*Caching, and other Patterns from the Literature* The patterns literature contains quite a few performance-related patterns relevant to implementing a *Model Extension*. If some values are very common for a particular property, *Flyweight* [17] might be justified. Beck discusses *Caching*, and observes that the biggest problem with caches is ensuring that they remain valid. To simplify this problem, he suggests reducing the scope and extent of the variable used for the cache. Thus, while it is usually quite easy to keep a caching temporary variable up to date, it is harder to do so for a caching instance variable [3, p. 44–5]. The variation that is most specifically relevant to us is the caching of *Model Extension* values *between client calls*. After all, the fastest way to compute something that hasn't changed since the last time we were asked is probably not to compute it at all, but to cache our last answer.

While a cache is certainly an important performance tool, caches can also be tricky in the best of cases, and costly in many. So let us examine some techniques for managing caches. Throughout, we assume that the implementation of each *Model Extension* keeps a cache of property values.

The third volume of *Pattern Oriented Software Architecture* [26] presents a set of patterns for managing resources efficiently. A cached *Model Extension*, containing as it does computed results, is not exactly the kind of resource assumed in this volume, but some of the patterns are still relevant. *Evictor* is a mechanism for ensuring that the elements of a cache are flushed at the appropriate time; it may be useful to us because we want to minimize the memory consumption of cached model extension data, and also prevent these data from being used when they are stale. However, *Evictor* gives us only the mechanism: how do we find an efficient policy for deciding which *Model Extension* to evict?

One way to avoid performing unnecessary analyses is to give the implementation of a *Model Extension* more information about the intentions of its clients. This is not a new idea, indeed, one of the advantages of monolithic programming tools is that usage information is available to their analysis components. However, once we have separated the tools from the model, we need to provide a way for the tools to pass usage information to the analyses. The pattern *Explicit Interest*, described on the next page, and its specialization *Life-long Interest*, described on page 30, show how to do this.

*Performance issues in the Running Example.* We use a *Model Extension* to give client tools a simple interface for obtaining the requirements of a class. Unfortunately, this simple interface makes it hard for the implementation to achieve good performance. For example, Schärli's algorithm is efficient when it is used to calculate the requirements of a class *and* its subclasses *at the same time*. What happens when a sequence of separate requests is made of the model? If the first class requested is the superclass of the others, then queries about the subclasses can be answered from the cache. However, if the tool happens to request subclasses before their superclass, the algorithm would repeatedly calculate the same information. A tool might try to work around this problem by applying the algorithm to some parent of the class in which it is really interested; however, this parent might have many subclasses in which the tool will never be interested, in which case this work would be wasted.

The root cause of the inefficiency is that the calculation mechanism does not know beforehand which classes the tool is going to enquire about. This situation is not specific to the required methods example. Consider the task of computing an *Inverse Mapping*: the most efficient procedure is to traverse all of the relevant code exacly once, gathering information about references to just those elements that clients will to ask about. We believe that the need to know in advance what will be interesting will be very common when performing a non-local code analysis.

One way of giving the calculation mechanism this extra information about usage would be to provide an additional "batch" interface. For example, we could provide an additional interface for required methods so that the client tool could

ask for the requirements of several classes at once. However, this interface would be inconsistent with the *Model Extension* pattern. Moreover, since the simple interface and the "high performance" interface would be quite different, the existence of the two interfaces would encourage premature optimization. Therefore we feel that it is better to keep the simple interface unchanged, and to add a separate interface by which the tool can explicitly express an interest in specific classes. The pattern *Explicit Interest* describes this additional interface, while *Batch Calculation* (p. 37) describes how to use it to obtain the same advantages as the additional "batch" interface that we just dismissed. *Minimal Calculation* (p. 32) examines when we can avoid updating the *Model Extension* at all, while *Eager Update* (p. 34), and *Lazy Update* (p. 35) address when to perform the updates that cannot be avoided.

---

## Explicit Interest

---

**Context.** You have a model extension that depends on calculations that are significant enough to require optimization.

**Problem.** How do client tools provide the information necessary to aid the optimizations?

**Forces.** The following forces are in play.

- Real-time display of the code and its properties requires interactive response times, which means 100 ms or better.
- Property calculations may be expensive and non-local, and the code model may be large.
- Caching all model extensions over the whole code model requires too much space, or re-validating such extensive caches after a change to the model requires too much computation, or both.
- The tools that are the clients of a model extension are focussed on only a small part of the whole code base at any given time.
- Having more information about the usage of a property can mean that we are better able to optimize the process of obtaining it.
- The code model is not completely static, because the programmer will occaisonally type in a change, or modify significant parts of the model by loading a new version of parts of the code.
- Any implementation tricks necessary to make the various model extensions fast enough should be hidden from client tools. The interface to each model extension should be as simple and uniform as possible

**Solution.** Add an interface to the model that allows clients to explicitly declare and retract their interest in a specific model extension for a specific code element. At any time, the parts of the *Shared Code Model* in which interests are declared are said to be *interesting*.

This solution is based on the assumption that, although caching the model extensions over the whole model is infeasible, caching them over the interesting parts of the code model is both feasible and cost-effective.

**Consequences.** Once this pattern has been applied, we may assume that tools will make queries on only the interesting elements of the code model. This assumption provides various opportunities for optimization. For example, caches can be focused on interesting information. This allows the client tools to assume that the space costs of caches are linear in the number of interests that they have registered.

Access to calculated properties of code elements not declared interesting can be either prohibited (useful to ensure that the interest declarations are complete), or merely less efficient (more friendly to tools before they adapt). This choice might also be controlled by a mode switch, to support different stages in the development of tools.

**Related patterns and variants.** *Explicit Interest* and *Observer* may seem similar because both make a model object aware of its clients. However, there are significant differences in the intent, effect, and consequences of the two patterns. An *Explicit Interest* is an optimization hint given to the provider of information by the consumer. This hint allows the provider more freedom of implementation; if the hint is not needed, it is always correct to ignore it. In contrast, adding an *Observer* creates the responsibility for the information provider to notify the new consumer of changes; this new responsibility can constrain the implementation, and cannot be ignored. For example, a requirement to include information about the new state in the notification message would force the calculation of that information before the message is sent. *Explicit Interest* has little consequence on the architecture of the application: declaring an interest does not affect when or how the consumer requests the value of the property. In contrast, *Observer* affects the control flow by placing initiative with the model, which must call the observer. The final difference is that with *Explicit Interest*, the model is not concerned with *who* expresses an interest, but solely with *which part* of the model is interesting. In contrast, *Observer* does say *who* is interested, but does *not* communicate which part of the model is interesting. In this sense, *Observer* and *Explicit Interest* are duals; they manage separate concerns, and can be used together.

Because the model is unconcerned with how many times an interest has been expressed, interests have some similarity to *reference counts* on the data structures supporting a model extension. As described by Meyers [29, Item 29],

reference counting is a pattern used at the application level to ensure that a data structure stays available for as long as it has at least one client. (Murray earlier outlined the same idea, under the name "use counts" [33, §3.2].)

*Explicit Interest* provides information that could be used by the other implementation patterns mentioned above. For example, *Lazy acquisition* might be applied only to non-interesting elements, and interest information could be used by an *Evictor*.

□

In applying *Explicit Interest* (p. 28) we decided that each instance of our code browser will tell the shared code model which classes it is currently displaying. Note that two browsers may display overlapping sets of classes, in which case there will be two registered interests in the requirements of those classes. Maintaining the interest registration requires the browser to declare interests when new classes are first displayed (for example, when a new browser is opened) and remove them later (when a browser is closed, or when the user changes the set of classes being viewed).

As is typical in large software development projects, we did not write this browser from scratch, but instead used an existing framework, the OmniBrowser[4]. The OmniBrowser is highly object-oriented: an OmniBrowser window does not display mere text strings, but model-specific *node* objects, each of which corresponds to (and references) a part of the code model, for example, a class or a method. These node objects have behavior, for example, they can react to mouse-clicks in useful ways.

In our browser, the node object representing a class has a life-span that matches quite precisely the interest of the browser in that class. Whenever a class is displayed, a node object corresponding to the class is created and included in the browser's display list; when the browser is no longer displaying that class, it ceases to reference the node object. We made the creation of a node object register an interest in the corresponding class; we also declare a finalization action, triggered when the node is garbage collected, that de-registers the interest. Thus, the requirements of a class are a *Life-long Interest* of the node that represents that class in our browser; this pattern is described below.

## Life-long Interest

**Context.** You have a tool that derives its extensibility from being an instance of a framework.

[4] http://www.wiresong.ca/OmniBrowser/

**Problem.** You wish to adapt this tool to present a new property. This property is captured by a *Model Extension*, so to use it efficiently, you need to express *Explicit Interest* in parts of the model being presented. How should you do this in a manner consistent with the tool framework?

**Forces.** The following forces are in play.

- If the framework pre-dates the publication of these patterns, it is unlikely to support *Explicit Interest* directly.
- To be effective, interests have to be declared before queries are made, and must eventually be retracted, though the timing for retraction is not critical.
- Failing to express an interest will hurt performance, possibly making the tool unusable.
- Extending a framework in a way that its designers did not anticipate is likely to produce code that is fragile and hard to understand.
- A well-designed object-oriented framework is likely to have *objects* representing the model elements that are interesting, and is likely to allow for customization by changing or replacing these objects.

**Solution.** Find the framework objects representing the parts of the model that enjoy the new property, and adapt them to express *Explicit Interest* in the corresponding part of the *Shared Code Model* for the *whole of their lifetime*. This can be achieved by making each of these objects register its interest when it is initialized or constructed, and retract its interest in when it is finalized or destroyed. A life-long interest can also be declared by modifying the factory that creates the representation objects. If we assume that the tool creates the representation objects before using the model extension, the interest declarations will be early enough; the language's object deallocation mechanisms will guarantee that interest is eventually retracted. This gives your objects the desired property, while assuming little about the framework, and making only local and easily understandable adaptations to the code.

**Consequences.** Life-long Interest is not always applicable: the framework may not have an object with an appropriate lifetime, the object may not be extensible, or its factory may not be available for modification. Using this pattern therefore constrains the implementation freedom of the framework. For example, caching the framework objects, rather than deallocating and reallocating them, will interfere with the use of this pattern.

The exact moment when a declared interest is retracted depends on when the run-time environment runs the garbage collector, and even on the nature of the garbage-colleciton algorithm.

**Related patterns and variants.** The C++ idiom *Resource Acquisition is Initialization* (RAII) can be used to implement this pattern. The C++ constructor will express an interest, and the destructor will retract it [22]. The immediate retraction that is the hallmark of RAII is not needed to implement *Life-long Interest*, but it does no harm.

□

So far we have discussed how to design an *interface* that encapsulates a code analyis as a model extension, while providing it with information about client intentions. We now resume the discussion of *implementing* a code analysis efficiently in the context created by these patterns.

Returning to our running example, it is clear that the naïve approach of re-calculating the requirements of every class on every request is far too expensive. In order to make calculations efficient, we look more closely at *what* to calculate, and *how* and *when* to do so. These three questions are not completely separable; we present patterns to address each, but they interact and the boundaries between them are not always clear.

*Minimal Calculation*, described below, is about limiting what we calculate. *Eager Update*, described on page 34, and *Lazy Update*, described on page 35, are about when to perform the calculation; each has its advantages and disadvantages. *Batch Calculation*, described on page 37, tells us when and how to perform the calculation of a non-local property that benefits from economies of scale: not as eagerly as *Eager Update*, not as lazily as *Lazy Update*, and all at once.

## Minimal Calculation

**Context.** You have a *Model Extension*, and an interface through which clients express *Explicit Interest*. You are maintaining a cache to avoid re-calculating the extension.

**Problem.** When the *Shared Code Model* changes, how do you avoid unnecessary re-computation of the cached properties in the *Model Extension*?

**Forces.** The following forces are in play.

- The code model is large, and ordinary editing operations change only a small part of it.
- However, this is not the case when changes to the code model are mechanized, as happens when a new package is loaded into the development environment, or a global refactoring (such as re-organizing the class hierarchy) is performed. In these situations, many parts of the model can change at the same time.

– At any particular time, only a small part of the code model affects the user's display.

**Solution.** Update only those model extensions that are both interesting *and* affected by changes in the code. The first criterion is applied by consulting the *Explicit Interests*. The latter is calculated using notification of changes to the code model and a knowledge of the dependencies of the property represented by the *Model Extension*.

**Consequences.** Implicit in the use of this pattern is that queries about *uninteresting* elements of the *Model Extension* can be answered slowly (or rejected outright). This should not be a problem, because these elements should not be queried often: that is the purpose of requiring a declaration of *Explicit Interest*. However, correctness requires that a *Model Extension* employing this pattern must invalidate cached information that is no longer interesting, rather than letting it go stale. (The invalidation might be lazy, that is, it could be done when the cache is queried).

If computing the update is not too expensive, and the frequency of calculations have been sufficiently reduced by the the use of this pattern, then it may be perfectly satisfactory to use *Eager Update*, described on the next page, to actually perform the computation as as soon as a change is notified. This will often be the case when the first force applies.

When the second force applies, this pattern can help to avoid slowing down the important work (the package load or the refactoring) by not wasting computation on updates to the code model that will not affect the tools that are currently in use.

**Related patterns and variants.** Even after employing this pattern, the amount of computation required to recalculate a global property may still be great enough that using *Eager Update*, described on the following page, slows down the important work. Moreover, it does so needlessly, because it repeatedly recalculates properties that will soon be invalidated. In these situations, consider using *Lazy Update* on page 35. If the computation exhibits economies of scale, also consider *Batch Calculation* on page 37.

□

When we have to recalculate the value of an interesting *Model Extension*, we can choose to do the work as soon as possible, or we can try to defer it as long as possible. *Eager Update*, described on the next page, characterizes the first situation, and *Lazy Update*, described on page 35, characterizes the second. Which is the better choice depends on the forces that apply and their relative strengths.

## Eager Update

**Context.** You have defined a *Model Extension* (p. 13) or a *Layered Extension* (p. 24) on top of a *Shared Code Model* (p. 9).

**Problem.** When should the properties that are captured by the *Model Extension* be recalculated?

**Forces.** The following forces are in play.

- The *Model Extension* is an *Observer* of parts of the *Shared Code Model*, and is notified when the relevant parts of the model change. Alternatively,
- the *Layered Extension* is an *Observer* of a single, lower-layer *Model Extension*.
- Recalculating the property is a fast, local operation.
- Most of the time, the user wants interactive response. However, bulk changes to the code, such as applying a patch that updates 100 classes, should not be interactive because this will needlessly delay the bulk operation.

**Solution.** Update the *Model Extension* eagerly, that is, as soon as you are notified of the change to the *Shared Code Model* or the lower-layer *Model Extension* on which it depends.

**Consequences.** Eagerness has several benefits.

1. The code is simple and readable: performing the update calculation is the simplest and most obviously correct response to a change notification.
2. The *Model Extension* will always be up-to-date, and thus client queries can be answered instantly.
3. *Eager Update* supports notification of changes to any tool or *Layered Extension* that depends on your *Model Extension*.

However, this pattern also has some disadvantages.

1. In the case of a bulk change to the code model, this pattern can be inefficient, because bulk changes may trigger many intermediate updates to the *Model Extensions*. These updates may be useless because they are almost immediately overwritten, or because the relevant part of the model becomes uninteresting before the user has the opportunity to query them.
2. A *Layered Extension* must not be recomputed until after the update of the lower-layer *Model Extension* on which it depends. However, if the *Layered*

*Extension* is also dependent on the *Shared Code Model* directly, then it may receive a change notification from the model before this update has completed. This can be a source of subtle and hard to find bugs.

□

*Eager Update* earns its place in the pattern language not because it is says anything surprising or unexpected, but because it supplies a name for the simplest thing that could possibly work. In particular, it provides a baseline to contrast with *Lazy Update*, described below.

## Lazy Update

**Context.** You have defined a *Model Extension* (p. 13) or a *Layered Extension* (p. 24) on top of a *Shared Code Model* (p. 9). You are maintaining a cache to avoid re-calculating the extension.

**Problem.** When, and in what order, should the cached properties that are captured by the *Model Extension* be recalculated?

**Forces.** The forces are similar to those that apply to *Eager Update*.

- The *Model Extension* may depend on other *Model Extensions* as well as on the *Shared Code Model*, and may be an *Observer* of all of them.
- The *Model Extension* will be notified when the relevant parts of the model change. However, the order in which the notifications arrive may be undetermined.
- The interdependencies between properties may be complex. The order in which the various *Model Extensions* are updated should be consistent with the dependencies.
- Recalculating the properties is a fast, local operation.
- Most of the time, the user wants interactive response. However, bulk changes to the code, such as applying a patch that updates 100 classes, should not be interactive because this will needlessly delay the bulk operation.

**Solution.** The *Model Extension* keeps track of all the relevant changes to the code model, but does not recompute its cached properties. Instead, it simply invalidates its cache. The *Model Extension* is updated lazily, that is, only in response to client queries; the new values are cached at that time.

**Consequences.** Laziness has several benefits.

- Model extensions that are not needed are not calculated.
- Batched changes will be executed without useless intermediate updates to the model extensions. The eventual update can be made more efficient using *Batch Calculation* (p. 37).
- Assuming that the dependencies between different *Layered Extensions* are not cyclic, their calculation will be performed in an appropriate order. Note that it is perfectly acceptable for the value of a property on one part of the code model to depend on the *same* property on a different part of the code model, even if both are maintained by the same *Model Extension*.

However, this pattern also has some disadvantages.

- The first access to a *Model Extension* after an update will take more time than other accesses. This may be visible to the programmer at the user interface.
- Perhaps the most significant disadvantage, in comparison with *Eager Update*, is that *Lazy Update* does not support change notifications on the model extensions.

**Known Uses.** In our running example, we used *Lazy Update* to minimize the set of classes for which we recalculate the required methods.

**Related patterns and variants.** When used in combination with *Batch Calculation*, this pattern is not completely lazy, so specific parts of a *Model Extension* may be calculated in spite of not yet having been requested. However, this is a reasonable tradeoff as long as the interests declared by the tools are sufficiently precise.

The pattern *Lazy Acquisition*, also from *Pattern Oriented Software Architecture — Volume 3* [26], suggests that delaying the moment of requesting a resource may improve performance, because sometimes the resource will not be used after all. *Lazy Update* can been seen as an application of this pattern, under the assumption that the client of the *Model Extension* does not actually request the resource until it is essential.

□

We now turn our attention to performing the calculation of a property as efficiently as possible. The required methods property is non-local — for example, a method selector may be self-sent by some superclass, making the corresponding method required; to find such requirements we need to check the code of all superclasses. This is reflected in Schärli's algorithm, which examines a class before its subclasses. These non-local aspects of the requirements calculation make it attractive to update the model extension in a *Batch Calculation*, described on the facing page, rather than one class at a time.

## Batch Calculation

**Context.** You have defined a *Model Extension* (p. 13) that depends on non-local aspects of the *Shared Code Model* (p. 9). For example, the value of a property for a class depends on the properties of all of its superclasses. Re-calculating this non-local property will therefore benefit from economies of scale. The interface to the model extension allows clients to express *Explicit Interest* (p. 28) in parts of the model.

**Problem.** How can you organize the recalculation of the property to take advantage of these economies of scale? In other words, how can you avoid repeated re-calculation of the *Model Extension*?

**Forces.** The following forces are in play.

- The properties upon which the *Model Extension* depends may change severally and repeatedly before the properties of the model extension itself are requested.
- The client of the *Model Extension* may request the extension's properties for multiple classes in succession.
- On the one hand, deferring updates of the properties until their values are requested, and then naïvely satisfying these requests, would result in multiple traversals of the common parts of the model on which they depend.
- On the other hand, recalculating the properties eagerly means calculating properties that may never be used.

**Solution.** Each *Model Extension* keeps track of all the relevant changes to the code model, but defers acting on them. When the model extension eventually updates its caches of calculated information, *all* changes are dealt with, and the extension properties are calculated for *all* interesting code elements, all at the same time. This makes it possible to use each traversal optimally.

**Consequences.** *Batch Calculation* reduces the number of traversals of the common parts of the model. However, it also means that for a period of time, the properties maintained by a *Model Extension* are invalid. Steps must therefore be taken to prevent clients seeing this invalid data.

This pattern tells us to defer acting on updates, but does not tell us when to stop procrastinating and get to work!

**Known Uses.** We applied this pattern in our implementation of the requirements browser for Squeak. *Batch Calculation* allowed us to use Schärli's algorithm as it was designed to be used: to compute in one pass a batch of requirements for classes that have an inheritance relationship.

**Related patterns and variants.** The timing and ordering of the deferred recalculation may be determined by *Lazy Update*, described on page 35. When this pattern is used in combination with *Lazy Update*, the computation of the *Model Extension* will not be completely lazy. Specific parts of the *Model Extension* may be calculated earlier than necessary, that is, before they are actually requested by a client. However, this is a reasonable tradeoff. As long as the interests declared by the tools are sufficiently precise, it is likely that those parts of the *Model Extension* that are calculated earlier than necessary will be requested by a client in the near future. Moreover, the economies of scale that accrue from *Batch Calculation* are likely to outweigh any inefficiency caused by computing a few properties that are never used.

□

The problem of displaying code properties maintained by a *Model Extension* is incremental in two different ways. First, requests for the property values are made incrementally. Second, changes to the code model are often small and spread over time.

Consider the timing of the recalculation of a property. *Eager Update* performs the calculation as early as possible: as soon as the new model elements on which it depends have been established. *Lazy Update* performs the calculation as late as possible: if it were left any later, a client request could not be answered. Between these extremes there may be a significant interval of time: *Batch Calculation* performs the recalculation somewhere in this interval. In its laziest form, it defers a batch of updates until the effects of *one* of them must be considered in order to satisfy a client request; at that time it eagerly processes the whole batch. A more eager alternative might be to wait until the CPU is idle, and then process the whole batch, ahead of the first client request.

When we implemented the requirements browser for Squeak, we used *Lazy Update* to order the updating of the *required methods* *Layered Extension* with the updating of the *self senders of* *Model Extension* that it uses. In its turn, the implementation of *self senders of* uses a combination of *Batch Calculation* and *Lazy Update* — the cache for a class is invalidated when the class is modified, and recalculated when it is requested. *Lazy Update* restricts the re-computation of *self senders of* to the specific class requested, rather than all interesting classes. This is sufficient for this particular model extension because the *self senders of* mapping is local to each class.

The complex interactions that result from these optimization patterns should cause you to delay optimization until you know that you need to do it, and then to reflect on the issues of correctness mentioned in the next subsection.

### 3.4 Correctness Concerns

As performance concerns drive the code implementing a model extension towards greater complexity, the code becomes more difficult to understand, and it becomes harder to avoid inserting bugs during maintenance and revision. How can we remain confident in the correctness of the implementation? Our answer is to test it against a *Canonical Implementation*, described below.

---

## Canonical Implementation

---

**Context.** You have implemented a useful model extension. You believe that the definition of the extension will be obvious to programmers familiar with the language supported by your environment, but the simple and obvious implementation based on the definition is not fast enough.

**Problem.** How do you improve performance while remaining confident of correctness?

**Forces.** The following forces are in play.

- Programmers normally have a good mental model of their programming language, so definitions rooted in that model are easy for them to understand.
- The calculation of a model extension must be fast enough for interactive use; this often necessitates optimizations that make the code deviate from that mental model, add complexity, and make it harder to verify.
- The model extension must provide correct information if users and tool builders are to trust it.
- Hand-written unit tests check only what their authors thought of testing.

**Solution.** Before proceeding to complicate the implementation with optimizations, take the simplest possible implementation and encapsulate it as the *Canonical Implementation*. Now you can freely create an *independent* implementation with better performance; this is the implementation that will actually be used by client tools. Write tests that compare the results of the two implementations over large existing code bases to gain confidence in the optimized implementation.

**Consequences.** Tests comparing the two implementations complement hand-built unit tests, because the data over which the tests run are independent of the assumptions in the efficient implementation.

For this pattern to be useful, the *Canonical Implementation* should be more likely to be correct than the optimized implementation, and should be more likely to *stay* correct. Why should these things be so?

1. Performance is not a concern for the Canonical Implementation, so you can use well-known, high-level libraries instead of hand-tuned code.
2. The Canonical Implementation need not read from or maintain any caches.
3. The Canonical Implementation can make use of data objects that support the semantics of the desired mathematical operations (*e.g.,* sets) rather than efficient ones (*e.g.,* arrays).
4. The *Canonical Implementation* is used only a test oracle for the fast implementation. This puts fewer constraints on its interface, so it can correspond more closely to a *Formal Definition* (p. 41).
5. You might choose to write the canonical implementation in a higher-level or more appropriate programming language than that chosen for the fast implementation.
6. The canonical implementation is not modified to meet performance or other pragmatic requirements, but only to fix bugs or follow changes in the formal definition. Therefore its code will change much more slowly, and bugs will be introduced into it less frequently, than will be the case for the fast implementation.

**Implementation.** Realistically, the need for this pattern will not become apparent until after some optimizations have already been applied, and the cost of debugging them has started to show. Thus, finding a good canonical implementation might require using version control to retrieve the simplest version that ever existed, and simplifying it a bit further.

Sometimes the course of development provides you with two different implementations, even though neither of them is canonical. This happened to us when collecting the self-sent messages of a method. The obvious implementation was to analyze the source code, which turned out to be so slow that it was necessary to cache the set of self-sent messages for every method in the system, and to rebuild the cache for a method whenever it was re-compiled. This in turn was found to use excessive memory, so the cache was cleverly compressed.

Some time later, one of us realized[5] that it ought to be possible to obtain the self-send information from the compiled byte-code using abstract interpretation. While debugging the abstract interpreter, we ran tests over *every method in the system*, comparing the results from the abstract interpreter to the cached results from the most recent compilation. We eventually arrived at a situation where there was a single method on which the two implementations differed — and concluded that the implementation that parsed the source code was less accurate. For this reason we hesitate to give it the name "canonical implementation."

[5] Following a conversation with John Brant.

**Related patterns and variants.** A canonical implementation can help you maintain confidence in the correctness of the optimized version. Sometimes a *Formal Definition* is also needed, in which case the canonical implementation can act as a bridge between the non-executable, but maximally clear, formal definition, and the efficient implementation used in practice.

□

While using the *Minimal Calculation* pattern, we came to believe that we could run the *requirements* algorithm less frequently if we took into account which classes had been modified and which classes implemented or self-sent each method selector. However, we found it difficult to be certain of the correctness of this optimization. What we needed was to prove a claim of the form: "if class $C$ requires a method named $s$, then one of the following statements must be true about the code..."

Proving this kind of theorem would be possible only if we had a formal definition of the *requirements* property. Some relevant formal definitions already existed [11] but were not particularly well-suited to our task. We found it useful to create a minimal *Formal Definition* based on just the concepts relevant to the requirements calculation: method lookup, explicit requirements, reachability through self- and super-sends, and required selectors. We used this definition to prove some necessary conditions for a message selector to be a *requirement*. In particular, we proved that if a selector is not defined in a class, not self-sent in the class, and not a requirement of its superclass, then it cannot be a requirement of the class.

These proofs allowed us to run the requirements extraction algorithm only when the necessary conditions hold. Because these conditions are cheap to check, and hold only rarely, performance was improved significantly, because we ran the costly algorithm much less often. This process is captured in the pattern *Formal Definition*, described below.

## Formal Definition

**Context.** You have thought of a property that is useful, but complex.

**Problem.** How can you be sure that the property is well-defined in all cases? How can you figure out what implementation shortcuts are possible, and convince yourself that they are correct?

**Forces.** The following forces are in play.

- The programming language that your IDE supports includes baroque features and corner cases that are rarely encountered in ordinary programs, but

which are nevertheless represented in the *Shared Code Model* and over which your property must be defined.
- Informal language is often imprecise when defining a property in such corner cases.
- To improve performance, you will want to refrain from examining parts of the program that cannot affect the value of the property. This implies that you need a way to be sure that a part of the program is *always* irrelevant to the property of interest.
- There may already be a formal system describing part of the language, such as a grammar, or a type system, or a formal semantics.

**Solution.** Use mathematical language — for example, sets, relations, and constraints — to define the property formally, in terms of primitive facts about the programming language and simpler properties. If possible, base your formal definition on an existing formalization of the programming language. When an optimization relies on a non-trivial claim about the property, *prove* the claim from the formal definition.

**Consequences.** Although it is still possible that the proof is incorrect and the optimization introduces a bug, the probability of this has been reduced. Moreover, unit testing of the optimized algorithm is likely to expose such a bug early, because the formal definition specifies the test result.

**Related patterns and variants.** The *Formal Definition*, translated into your programming language of choice without regard for efficiency, can become a *Canonical Implementation* (p. 39). For example, the canonical implementation might be written in terms of sets and relations in a functional style that mimics the mathematics.

□

## 4 Pattern Language Summary

We conclude our description of the pattern language with a quick overview of our solution to the problem of building a extensible, modular architecture for representing a program in Squeak Smalltalk.

One important property of the Smalltalk programming environment is that it has a *Shared Code Model* (p. 9) on which we could build. Since the shared code model does not maintain the required methods of a class, we implemented a *Model Extension* (p. 13) that exposes the required methods as if they were part of the code model. We realized that the Squeak shared code model is not minimal, but in fact includes an *Alternative Representation* (p. 20) for methods.

Calculating the required methods for every class in a large application would be prohibitively expensive, and much of the effort would be wasted because programmers are interested in studying only a few classes at a time. The model

extension therefore allows tools to express *Explicit Interest* (p. 28) in the properties of a specific class.

In the browser development framework in which we were working, we found that a simple way of adapting the browser to express *Explicit Interest* was *Lifelong Interest* (p. 30), in which a particular object's interest endures until it is garbage collected. Knowledge of the "interesting" classes creates a context in which various optimization strategies are applicable; two optimizations that we consider are *Minimal Calculation* (p. 32) and *Batch Calculation* (p. 37). *Lazy Update* (p. 35) complements them by determining when recalculation of a property should take place after a model change.

To prevent this preoccupation with efficiency from coming at the expense of understandability and correctness, we used a *Formal Definition* (p. 41) and a *Canonical Implementation* (p. 39) as a test oracle. We found that the (rather complicated) requirements property depends on two simpler properties, which led us to *Layered Extension* (p. 24). One of those properties turns out to be useful both as an intermediate layer for a higher-level calculation and also to the end user. It is an *Inverse Mapping* (p. 22), and as such exemplifies a class of properties that are frequently useful to programmers using the IDE, to analysts, to architects building more complex extensions, and to toolsmiths.

These patterns make it easier to write a second tool that uses an existing analysis, and also make it easier to adapt an existing tool to make use of a new analysis. *Generic Tools* (p. 17) represent the limit of this second case — tools designed to make use of any property of the code model exposed by an extension, and thus to lower the barrier to using a new analysis.

## 5 Review of the Running Example

A pattern language is useful if it leads to an improvement in the architecture, functionality, performance or reusability of software that adopts it. In this section we return to the Requirements Browser example, and consider how the application of this pattern language has improved it as a product.

The original version of the Requirements Browser [39] was implemented as part of an incremental programming environment. One of the principal goals of this environment is to show the programmer in real time the actual state of the code being developed; this includes what methods are still missing, and which classes are incomplete. Meeting this goal demands responsiveness during typical browsing activities, and the constant display of requirements information. These are difficult demands to satisfy, because the requirements calculation is non-local, and potentially quite expensive.

As we mentioned on page 40, in the initial prototype, all of the self-send information for every method in the image was calculated eagerly and cached in a compressed form. The super- and class-send information was also cached, adding to the space cost. This cache was subsequently replaced by a custom abstract interpreter that computed the send information from the bytecode on demand. However, achieving responsiveness still required that the *Inverse Mapping* *self*

*senders of* be cached. This cache, and a global cache of requirements information for every class in the system, were updated eagerly whenever a method changed. What were the performance and deployment implications of these caches?

Our measurements showed that the total memory footprint of these caches was around 2 MB, for a code model (class objects and bytecoded methods) of 4 MB. The cache was updated at every change of any method. This worked reasonably well for interactive changes to single methods, but negatively affected bulk recompilations, such as those caused by loading a new version of a package. This was true even if the package being loaded had no effect on the requirements being displayed. Building this cache from scratch, as was required to deploy the requirements browser on a Squeak image for the first time, took tens of minutes.

The patterns described in this article made it easier to overcome these problems. By caching information only for those classes in which there was *Explicit Interest*, we reduced the cache size to be proportional to the amount of code being displayed, rather than the amount of code loaded in the system. *Lazy Update* removed any need for long re-computations when installing the system, and speeded up bulk package loads. Some of the optimization required to make the incremental computations efficient were quite complex, but *Canonical Implementation* and *Formal Definition* greatly increased our confidence in their correctness.

The original Requirements Browser prototype added the implementation of the *required methods* property directly to the *Shared Code Model*; the use of a *Model Extension* allowed us to avoid this modification of the core system classes, which had proved to be a packaging and maintenance problem.

Applying the pattern language presented in this article thus made available for reuse a complex code analysis that was originally buried in a particular tool. At the same time, the analysis became more practical from the point of view of performance, and less intrusive in the code model. We note that while the particular example that we chose — abstract classes — is commonly part of the explicit code model in statically typed languages, global analyses and the enhanced models that they require are not specific to latently typed languages.

## 6 Historical context

The idea of multiple-view software development environments has been studied at least since 1986 [19], when David Garlan published the work that led to his doctoral thesis [20]. The Field environment constructed at Brown University in the early 1990s by Steve Reiss and his students was a landmark in the development of such systems. A 1992 study by Meyers and Reiss [31] examined novice users of Field and concluded that multiple views, or at least the particular set of views that Field supported, did indeed help programmers to perform maintenance tasks.

However, Field was constructed as a loose collection of separate tools that communicated using what we would now call a publish and subscribe system (Meyers called it “Selective Broadcast” [28]). Although this made it quite easy to write new tools and add them to Field, each tool duplicated the core data

of the system, making it hard to maintain consistency, contributing to high latency when attempting to keep simultaneous views up-to-date, and inevitably forcing programmers to introduce redundancy between the tools. The approach to consistency that we are taking in Multiview is close to what Meyers called "Canonical representation", which seemed then to be an unattainable dream.

Since 1991, the amount of core memory available on a typical development workstation has expanded from 16 MB to 2 GB. This has made it possible to keep all or most of the representation of even quite large software systems in core memory, and this permits the use of more flexible data structures than are supported by a database and, perhaps more importantly, allows the parts of these data structures to link to each other directly. Nevertheless, it is still the case that "no representation has yet been devised that is suitable for all possible tools". The idea of an *extensible* architecture for code models and the pattern language described in this article is a response to the (belated) recognition that no such representation will ever be devised *a priori*.

From a review of previous research, Meyers concludes that a Canonical Representation based on abstract syntax trees (ASTs) is insufficient. Marlin [27] presents an architecture (also called MultiView, but written in CamelCase) that takes this approach, and concludes that at least part of the problem is that the AST "shows through" in the form of the syntax-structured editor. The hybrid style of Smalltalk's *Shared Code Model* avoids this difficulty by representing method bodies as text. Experience has shown that textual views have advantages over structured views at the method-level: textual views keep white space and indentation which, while semantically useless, are important documentation [43]. Editing text also makes it easier for the programmer to transform the code by permitting it to pass through syntactically illegal intermediate states.

Meyers and Reiss [30] describe another problem with ASTs: they do a poor job of supporting views that are unrelated to the program's syntax. In their search for a "single canonical representation for software systems", they present their own Semantic Program Graphs (SPGs), which support a range of views directly. Meyers and Reiss themselves note that SPGs do not faithfully represent all aspects of a program; one of their solutions is to allow clients to annotate them.

The architecture that we propose in this article combines the advantages of a Canonical Representation with those of multiple specialized representations connected by message passing. The *Shared Code Model* solves consistency problems by being the unique modifiable representation, but additional representations can be made available (as *Model Extension*s) to help support disparate views. Thus, the research into advanced representations such as SPGs can be leveraged by using these representations as model extensions.

## 7 Coda

It used to be that conducting research in program development tools required either settling for a mediocre user interface (making it unlikely that the ex-

perimental tool would be widely adopted) or creating an environment in which to embed the tool, a larger investment than most research projects could support. Fortunately, extensible development environments, such as Eclipse, are now available: these environments are creating ecosystems in which IDE architects, toolsmiths and program analysts can support each others' work. In such an IDE, the investment required to move from the idea for an analysis to a usable tool is reduced. Because these environments are widely used, tools embedded in them are more likely to be used than standalone tools. The extensibility of these IDEs make it possible for them to include capabilities beyond their architects' initial planning and priorities; if users are likely to find a tool useful, sooner or later the toolsmiths will respond to that demand.

The patterns presented in this article are intended to facilitate this process. A development environment is made extensible by the frameworks it provides and by the idioms it promotes for sharing code between extenders. The patterns that we have described support the use of a *Shared Code Model* to which code analyses can be added systematically. These analyses can then be shared between different extenders of the development environment, so that not only tools but other analyses can build on them.

The early patterns, particularly *Shared Code Model* and *Model Extension*, are known to exist in Squeak, and in some form in Eclipse; we do not know how widespread they are. Some of the later patterns may not have been used other than in our own tools. Thus the architecture that we propose must be considered preliminary, and it seems likely that some of the proposed patterns will change, and possibly more will be added, as more architects, analysts and toolsmiths gain experience with it.

Our confidence that these patterns will be found useful is based on the same reasoning that argues for the usefulness of the object-oriented paradigm itself. Objects facilitate reuse when they represent entities in the real world. As a consequence, different applications that manipulate the same concerns in the real world are likely to be able to reuse the corresponding objects in an object-oriented design. Of course, these objects may need to be extended — but this is also something that the paradigm supports.

In a program development environment, the "real world" that is the meeting place for the various tools and analyses is the program under development. The representation of that program — the *Shared Code Model* — is highly reusable because every tool builder and analyst understands the subject programming language and the properties of its elements. For example, anyone who knows Java would assume that a code-model element that represents a Java class will provide a method or attribute to access its superclass — and they would be right to do so. In contrast, properties that are available only by running an analysis and building a separate result graph, which the client must then worry about keeping up to date, are much less likely to be reused.

We are aware that there are some flaws in this argument. While many analyses do indeed create new properties of existing elements in the *Shared Code Model*, there are code analyses in the literature that create new entities that are

not local to any such element. One example is finding cyclicly dependent sets of classes [42]. While the results of this analysis can be mapped back on to the *Shared Code Model*, that is not the "obvious" way of looking at the them. Also, the tradeoff between *Lazy Update* and *Eager Update* is worthy of more study; we advocated *Lazy Update* for performance, but *Eager Update* has the benefit of allowing a element of a *Model Extension* to be *observable*.

In spite of these issues, we feel that we have made enough progress with this architecture to expose it to the scrutiny of the programming environment community. A demonstration of our implementation is available as a Squeak image at http://www.cs.pdx.edu/~black/goodies/TraitsBeta3.0/TraitsBeta3.02.tgz. It has begun to solve the very real problem first identified by Meyers in 1991 [28], and has done so in a way that enables us to build useful tools for Squeak. We hope that others will be encouraged to critique and expand on these patterns, and to report their findings.

## Acknowledgments

This work was partially supported by the National Science Foundation of the United States under awards CCF-0313401 and CCCF-0520346. We also thank Emerson Murphy-Hill and Philip Quitslund for motivational discussions, John Brant for help in building the abstract interpreter, and Colin Putney for his willingness to adapt the OmniBrowser to our needs.

We are indebted to an anonymous OOPSLA referee for information about the Cadillac system, to our PLOP 2006 shepherd Peter Sommerlad for his extensive advise and useful comments, and to the many members of our PLOP writers workshop, particularly Ademar Aguiar. The anonymous referees for this journal also provided numerous suggestions that have improved the article.

## References

1. Ken Auer and Kent Beck. Lazy optimization: patterns for efficient smalltalk programming. In *Pattern languages of program design 2*, pages 19–42. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
2. Kent Beck. Simple Smalltalk testing: With patterns. http://www.xprogramming.com/testfram.htm.
3. Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.
4. D.W. Binkley and K.B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances of Computing*, volume 43, pages 1–50. Academic Press, 1996.
5. Andrew P. Black and Mark P. Jones. The case for multiple views. In *Workshop on Directions in Software Engineering Environments, ICSE 2004*, pages 96–103, May 2004.
6. Andrew P. Black and Nathanael Schärli. Traits: Tools and methodology. In *Proceedings ICSE 2004*, pages 676–686, May 2004.
7. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

8. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.
9. Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.
10. Stéphane Ducasse and Michele Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.
11. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, March 2006.
12. Eclipse Foundation. JavaDoc page for Interface ITypeHierarchy. http://help.eclipse.org/help32/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/ITypeHierarchy.html.
13. Richard P. Gabriel. Lisp: Good news, bad news, how to win big. In *First European Conference on the Practical Applications of Lisp*, Cambridge University, Cambridge, England, 1990. (keynote address).
14. Richard P. Gabriel, Nickieben Bourbaki, Matthieu Devin, Patrick Dussud, David N. Gray, and Harlan B. Sexton. Foundations for a C++ programming environment. In *Proceeding of C++ at Work*, September 1990.
15. Erich Gamma. Extension object. In *Pattern languages of program design 3*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
16. Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
18. Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.
19. David Garlan. Views for tools in integrated environments. In *Proceedings of an International Workshop on Advanced Programming Environments*, pages 314–343, London, UK, 1986. Springer-Verlag.
20. David Barnard Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, January 1988.
21. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.
22. Jon Hanna. The RAII programming idiom. http://www.hackcraft.net/raii/, accessed January 2007.
23. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.
24. Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2001.
25. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

26. Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture Volume 3 – Patterns for Resource Management*. John Wiley and Sons, 2004.
27. Chris Marlin. Multiple views based on unparsing canonical representations — the MultiView architecture. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96)*, pages 222–226, New York, NY, USA, 1996. ACM Press.
28. Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Softw.*, 8(1):49–57, 1991.
29. Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
30. Scott Meyers and Steven P. Reiss. A system for multiparadigm development of software systems. In *IWSSD '91: Proceedings of the 6th international workshop on Software specification and design*, pages 202–209, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
31. Scott Meyers and Steven P. Reiss. An empirical study of multiple-view software development. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 47–57, New York, NY, USA, 1992. ACM Press.
32. Emerson Murphy-Hill. Improving refactoring with alternate program views. Technical Report TR-06-05, Portland State University, May 2006. http://multiview.cs.pdx.edu/publications/rpe.pdf.
33. Robert B. Murray. *C++ strategies and tactics*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1993.
34. Object Technology International, Inc. Eclipse platform technical overview, 2003. White paper.
35. Dirk Riehle and Heinz Züllighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. In *Pattern languages of program design 1*, pages 9–42. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
36. Linda Rising, editor. *Design Patterns in Communications Software*. Cambridge University Press, 2001.
37. Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, February 2005.
38. Nathanael Schärli and Andrew P. Black. A browser for incremental programming. Technical Report CSE-03-008, OGI School of Science & Engineering, Beaverton, Oregon, USA, April 2003.
39. Nathanael Schärli and Andrew P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 30:79–95, 2004.
40. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.
41. Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.
42. Daniel Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.
43. Michael L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, October 2002.
44. Roel Wuyts. Star Browser. http://homepages.ulb.ac.be/~rowuyts/StarBrowser/index.html, accessed May 2007.