A Pattern Language for Extensible Program Representation

– Id: ModelExtensions.tex,v 1.131 2006/09/25 07:33:39 black Exp –

Daniel Vainsencher
The Technion
danielv@techunix.technion.ac.il

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

September 25, 2006

Note to PLOP Participants

We are aware that this paper is too long to be considered in its entirety in a writers' workshop. We suggest that the workshop focus on the first 20 pages (up to the end of section 3.2) since this part is the most polished. This unfortunately omits Section 3.3, starting on 21, which in some ways is the heart of the paper. General feedback on the rest of section 3 would be welcome, but significant changes are already planned.

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 views 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 and optimized representation. This limits the usefulness of the analyses embedded in those representations, and also limits their reusability by other tool builders. Unintegrated tools also produce inconsistent views, which reduce the value of multiple views.

This paper describes an architecture that allows 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. We present architectural patterns that address efficiency, correctness and the integration of multiple analyses and tools in a modular fashion.

1 Introduction

We are trying to build better programming environments. Along the way, we think that we have discovered some interesting things about representing a program inside a modern programming environment. The purpose of this paper is to share these discoveries with you, in the form of a pattern language for representing programs.

Our view of programs is that they are complex, multi-dimensional structures, not linear text [5]. Our view of programming environments is that they are tools to reveal and elucidate that structure. This is not a new view. Even the most conservative of programmers now expects features such as syntax coloring and parenthesis matching; these are views that reveal and elucidate the fine-grained context-free syntactic structure of procedures, modules, methods and classes. Less conservative programmers also expect context-sensitive command completion and the ability to navigate from uses to definitions of a program component, and back again; these are views that expose some of the context-sensitive syntax of the program. However, we aim to push this point of view further than others have done, by making available a wide range of views of the program, some of which are derived from various kinds of program analysis. We also want to make it easy to extend our environment with new views, as a demand for them emerges. We are thus faced with a difficult engineering problem. Whose responsibility is it to perform the code analyses that the views require? Should each view be created and maintained by a dedicated tool that operates on a shared collection of text files, or on a shared program database? Or should there be a "universal" shared code model that is general enough to directly provide the data for all views? Such a shared model would need to make available not only all the information directly present in the code base, but also all of the indirect information that can be inferred from it, just in case some view might ask for it. This sounds difficult, but has the important advantage of ensuring that all of the views are consistent.

Our pattern language advocates such a shared code model. We are by no means the first to have done 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." [25].

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.

1.1 The Pattern Language

The main contribution of this paper 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 view maintainers? 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 to add difficult-to-find bugs; the fourth group of patterns (Section 3.4) is about removing them.

Index to patterns

The essence of the pattern language that we propose here is to apply the model-view architecture to program development environments. The code — *and all of the interesting analyses on it* — become the model; the various tools in the environment do nothing more than ask the model for the data that they need, using *Observer* in the usual way, and present it on the screen. We accommodate all of those interesting analyses in the model — including the ones that we haven't yet realized that we need — by making 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 model that a tool might ask, and must do so quickly, so that the view can respond in real-time as the code is modified. This is why we pay such attention to performance; without a solution to this challenge, our pattern language 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 that we have implemented, and then use it throughout the paper as a running example to demonstrate the issues and how the patterns address them. But first, we offer an apology.

1.2 Patterns or Proto-patterns?

In the introduction to Linda Rising's collection *Design Patterns in Communications Software* [30], 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. [30, p. xii].

While there is no one definition of what makes a design pattern, as Schmidt indicates, a useful rule of thumb is that patterns present a solution to a problem in a context. Another criterion is that patterns 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 architectures in a common domain and to open a dialog in and around it.

By all of these criteria, the pattern form is appropriate for this work. However, there is commonly also an expectation that a pattern distills from multiple experiences. For example, Buschmann *et al.* [8] propose finding at least 3 examples of an idea when pattern mining, and Gamma *et al.* [15] offer at least two examples of each pattern. By this criterion, the strategies that we propose do not yet qualify as patterns because we cannot offer evidence that they are currently in wide use. We have implemented these ideas in the context of the Smalltalk programming toolset, but have not yet seen most of them adopted in other environments, for example, Eclipse. However, we feel that presenting these *proto-patterns* at this stage will enable more development environments to build on this architecture in the future, and in the process extend and evolve our contributions into a full-fledged pattern language. 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 paper.

2 Motivating 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, 35]. 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:, or 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, which serve to mark the method as abstract. Collection

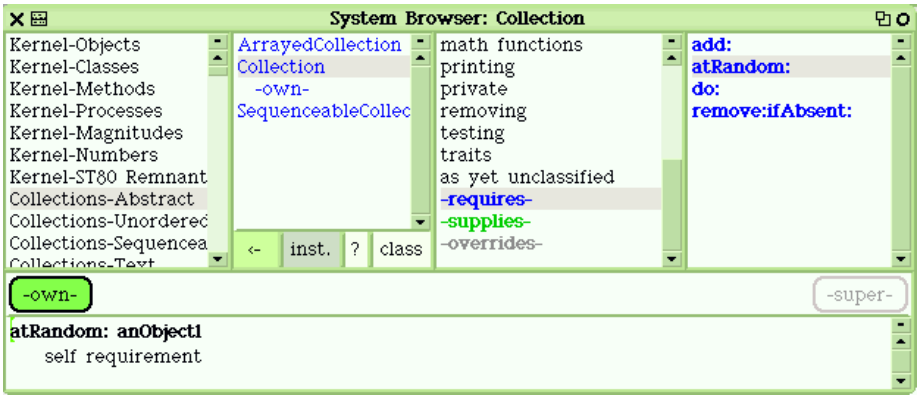

Figure 1: The Smalltalk browser showing abstract classes and required methods. In the method list pane (at the top, on the far right), all of the required methods of class Collection are listed. In the class pane (at the top, second pane from the left), abstract classes are highlighted in blue. The fact that they are abstract is deduced by the browser when it finds that they have a non-empty set of required methods.

does provide concrete methods for addAll:, remove:, collect:, select: *etc.*, which are 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 atRandom: is sent to self in another method, even though it is not explicitly marked as abstract: atRandom: is an implicit abstract method. To recap: 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 are required to make it concrete. 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 are known as its *requirements* [34]. It is particularly important to infer the implicit requirements because this supports "programming by intention" [19]: the constant display of the required methods acts as a "to do list" for the programmer. 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 [32, 34]. Our problem, then, was to construct from this algorithm a practical browser that would indicate which classes in a (pos-

sibly 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" we mean "make the information concrete". In a sense, the implicit information is there in the code model all of the time, but 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 that which 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

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 IDE tools that use it, and the interfaces needed to support this division. The basic concepts are a code model that is shared by different tools, and enriched over time by different model extensions. The value of new model extensions is increased by the presence of some generic tools that leverage them with little investment.

The second group (section 3.2) addresses the content of the extended model, by suggesting a number of categories into which model extensions may fall, and how extensions can build upon other extensions. The third group (section 3.3) provides guidance on making the extensible model fast enough while maintaining the cleanliness of the interface and correctness of implementation. The fourth and final group (section 3.4) addresses correctness. Figure 2 shows the relationships between the patterns and the problems that they address.

3.1 A Code Model supporting Multiple Views

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

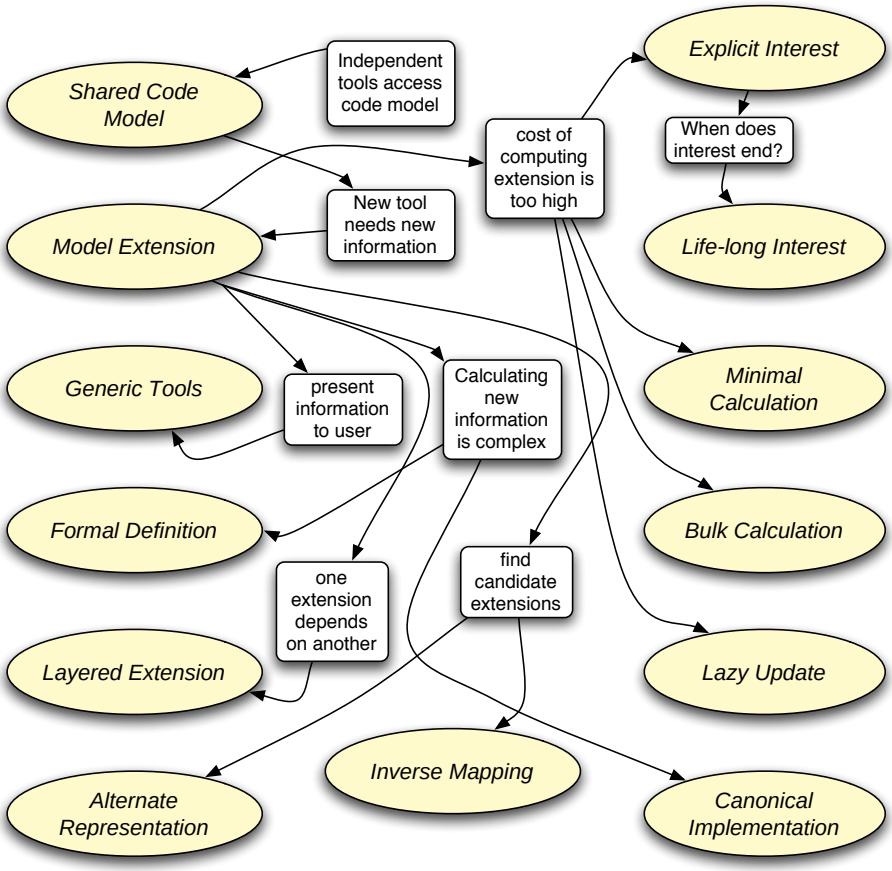

Figure 2: Text in rounded rectangles summarizes a problem. Shaded ellipses name patterns discussed in the body of the paper. Arrows show how problems are addressed by the patterns, and how the use of the patterns gives ride to new problems.

the project. In contrast, we implemented the requirements browser on top of Squeak Smalltalk [18], which uses a rather different code model, common in Smalltalk environments. 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 on the following page, 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 tools are written by different developers. Multiple representations of the code lead to high maintenance costs, and to inconsistencies between the tools. The model needs to be *complete* so that all tools can be based on it. The whole development environment needs to be responsive for common actions, so requests on the model should be sufficiently cheap. 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 examining 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 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* pattern. 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. The fact that the code model is shared and the use of *Observer* allows multiple tools to remain synchronized and oblivious to one another.

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.

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. Another is when to ignore the minimality rule, and store both. Sometimes there are good reasons to duplicate information, since some formats are more convenient for specific tasks. An obvious example is compiling source code into byte-coded methods to speed execution. The source and the byte-code are essentially two representations of the same information, and the code model must go to some lengths to keep them consistent.

It would be much simpler to execute the text directly, or to re-compile it whenever needed, and this is in fact what most Ruby and Python implementations do. An alternative way of achieving minimality would not to keep the source code in the model at all, and instead to re-generate it from the byte-code when necessary. However, discarding the source would also discard layout and comments — the so called "documentary structure" of the code [40], which is usually considered important for methods. We have seen that there exist cases in which some duplication is warranted inside the *Shared Code Model*. But in most cases it is not necessary; *Alternative Representation*, described on page 16, explains how multiple representations can be used conveniently, without making them part of the *Shared Code Model*.

Note that although Squeak does redundantly store both source text and byte code for methods, it does not preserve the source text when the documentary structure is deemed unimportant. For example, the text of class definitions, whose formatting is stylized, is not preserved.

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.

Related patterns and variants. Riehle *et al.* [29] present the Tools and Materials metaphor, which motivates the distinction between the application (a tool) and the model on which it operates (the materials). Other patterns such as MVC [8] have also made this distinction. In terms of the metaphor, our pattern language aims to improve the stock of readily available materials so that the work that must be performed on them using tools is reduced, or eliminated altogether. (The Tools and Materials metaphor is discussed further in Section 5: Related Work.)

We have already mentioned the role of the Observer pattern in connecting the code model to the tools that operate on it.

□

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 requiredMethods.

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 was

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

Thus, clients had responsibility of ensuring that the cache was up to date, which 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. The next pattern, *Model Extension*, deals with this trade-off.

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 the tools access properties that are not stored in the code model, but are calculated from it?

Forces. 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

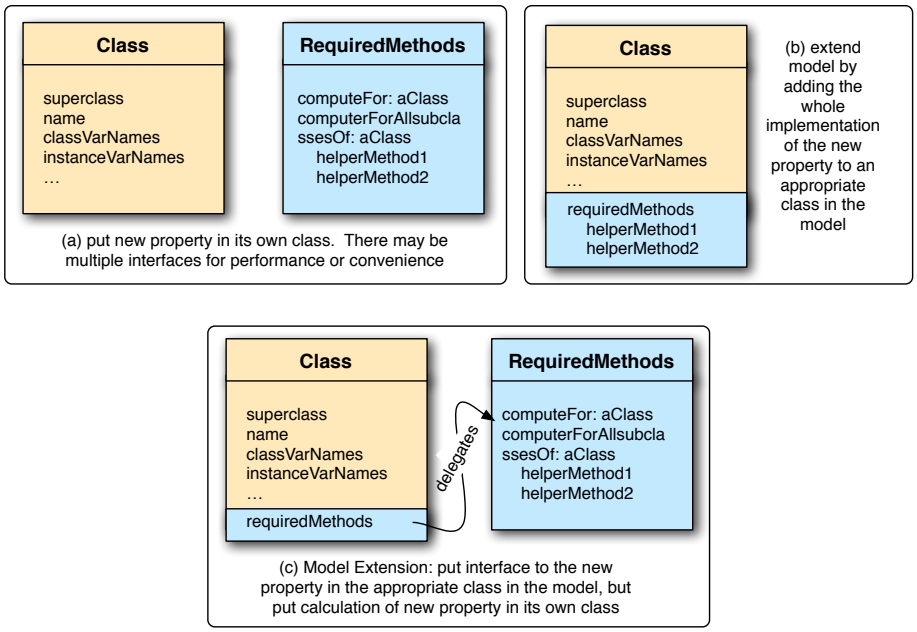

Figure 3: Here we show diagrammatically various alternatives for enriching a shared code mode. In (a), Class is part of the shared code model, shaded buff. The algorithm to compute the required methods of the class has been added in a separate class, shaded blue. The means that the user of each extension must be aware of a non-uniform interface to that extension, and may also have to be concerned with initializing and finalizing the classes that implement it.
In (b), the whole of the implementation of the extension had 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 demands that the programming language provide a very complete implementation of class extension.
Alternative (c) exploits the *Model Extension* (p. 10) pattern; the interface to the extension is in the appropriate code class, but the implementation is encapsulated in its own class. Only the most modest class extension mechanism is required.

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, multiple implementations must be avoided.

One of the ways in which a new tool 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 a tool may be complex, and should be encapsulated and kept separate 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.

In our running example, the interface that we desire is myClass requiredMethods, since this will make requirements accessible as directly as any other properties of the code model. To the tool writer, extensions like requiredMethods add richness to the otherwise minimal code model, making it a more useful representation of the code. By accessing *all* aspects of the code through the code model, we make the implementations of different tools simpler, and the tools more similar to each another. This makes it easier to use, understand and maintain different tools.

However, the implementation of the logic necessary to implement the new interface should be placed in its own class (or classes), and the extensions to the model classes should delegate all of their responsibilities to this new class. This arrangement is illustrated in Figure 3(c). For authors of model extensions, the separation of the implementation from the interface enables them to address the complexities of correct and efficient computation of each property without increasing the complexity of the code model.

The interface extensions should be *class extension* methods if the implementation language supports them, or Extension Objects [14] otherwise.

Consequences. Model extensions provide the flesh to the *Shared Code Model*'s skeleton. Tools can now push all responsibility for interpretations of the code model into extensions, where they are available to any new tools.

Many programming languages do not recognize the importance of *open classes* [9]; 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 change to the interface of a standard class library requires a change in its implementation, which is often not under the control of the tool writer. Some Java variants such as MultiJava [9] and AspectJ [20] recognize the value of open classes and do provide for class extensions; programmers using ordinary Java can fall back on the *Extension Object* pattern [14].

Smalltalk supports open classes, so the Smalltalk programmer can extend the interface of the code model by adding new methods in a separate code package containing a class extension. However, the Squeak class extension mechanism is not well-suited to modifying the structure of existing class definition; for example, it is impractical to change the representation of the code model by adding some instance variables to a class.

Because this pattern places all of the logic of the model extension in new classes, the only class extension facility that is needed is the ability to add a

new stateless method to an existing class. This mitigates the impact of any deficiency of the chosen implementation language.

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. This is important: 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*, which describes a simple interface addition that allows various implementation strategies, including the well known *Caching* [21].

□

Using the patterns that we have described, we implemented a *requirements browser* for Squeak, which flags abstract classes 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 that: it should be possible to make use of a model extension even without requiring that the environment 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 the programmer a list of all of the predicates on classes, and allow him or her to decide *which* predicate will determine class coloring. Then, 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 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, a new code analysis, if given a sufficiently abstract interface, provides us not 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*, described below.

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 as accessible as possible?

Forces. Many known and useful code analyses are understood by few practitioners and used by even fewer. This is true even though tools exist to apply them, because activating a specialized tool just to access an analysis takes time, and thus makes the analysis less valuable. If the programmer must learn a new tool in order to access a new analysis, he is less likely to ever 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, and provides a disincentive to making the analysis reusable.

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. There are sophisticated techniques for visualizing and manipulating this kind of data that are not dependent on their particular meanings.

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 to use information from different sources.

Note that the requirement here is extensibility by a programmer, which can be achieved at different levels by different means. An 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. Generic tools are therefore are more likely to complement than to replace more specialized tools.

Related patterns and variants.

We present two more examples of *Generic Tools*.

First, we observe that a unit testing framework such as SUnit [3] 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 desired 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: (self subclasses isEmpty and: [self 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 [41], 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 to do the same thing. The latter type of customization can be done 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 Differentiating Model Extensions

This section presents two general situations in which application of *Model Extension* (p. 10) 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 smart development environments. In each pattern we therefore mention the performance problems typical to each kind of model extension and reference 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 Abstract Syntax Tree representation would be more convenient.

Forces. We cannot change the shared code model representation to fit our application. Nevertheless, because the shared code model is complete, the information we need is in it somewhere: it is implicit where we would like it to be explicit.

Solution. Define the representation that you prefer as a model extension. 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* [21].

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 a 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, and explicit

in the alternative 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, *etc.*. Moreover, the ad-hoc solution would to some extent duplicate the normal parsing code that exists elsewhere. Applying this pattern allows us to avoid such duplication.

Related patterns and variants. It is worth noting that many alternative representations exist and are commonly used, and while they might be useful to programmers seeking to better understand their programs, these representations are rarely shown to them. Some of the representations used by compilers to optimize code include control flow graphs (CFG), single static assignment (SSA) form, various intermediate languages and finally the compiler's target language. While the programmer often works with higher-level representations, sometimes these low-level ones are of interest, for example when investigating the performance of specific bits of code. Compilers often allow the programmer to see at least part of the representations that they use, but 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.

□

A very common activity while reading code is navigating through references. 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-orinetd language, an IDE may provide a pop-up menu on a message-send that allows the programmer to choose between the different relevant method bodies. This functionality is not particularly difficulty to implement; it is enough to parse the particular method being browsed (possibly this would be an *Alternative Representation* (p. 16)) and keep track of relevant procedure and method definitions in the program. It isn't hard to keep this information current: when a procedure or a method is changed, the IDE needs to update the parse tree of that procedure or method only.

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. The solution that we propose is captured in the pattern *Inverse Mapping*, described on the next page.

Inverse Mapping

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

Problem. Can we allow efficient navigation in the opposite direction?

Forces. 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. Programmers and higher-level model extensions may both wish to use these mappings in the other direction. Traversing the code base to search for such references is expensive.

Solution. As we will see, efficient implementation of inverse mappings can be complex. Provide each inverse mapping as a *Model Extension*, so that the implementation details are hidden from clients.

The basic difficulty in the efficient implementation of an *Inverse Mapping* is that finding all references to a code model element is a non-local task, because 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 expensive in space for references that are common, such as procedure calls or message sends.

Assuming a cache is used, what does updating the cache for an *Inverse Mapping* entail? Suppose that the part of the code model that has been modified corresponds to a procedure definition. We have to remove from the cache those entries related 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 the new one. If this is not the case, it may be necessary to keep a separate record of all references in the ordinary direction, just for this purpose, or, in the worst case, to rebuild the cache from scratch by exhaustive search.

Some ways to reduce the cost of providing an *Inverse Mapping* in this case are described in the efficiency patterns in Section 3.3.

An 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. So 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.

The Smalltalk environment does not presently adopt this pattern. Indeed, it the base system does not put *either* enquiry in the code model; instead the functionality to find all references to a class is implemented in the SystemNavigation tool (using global search). 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 cost space (for a cache of an inverted index).

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 with information only about references to specific code model elements. *Bulk Calculation* and *Lazy Update* show how to make the most of each non-local scan.

□

Calculating the inverse mappings we mentioned above, we assumed that the references made in a method are available. In turn, these might be found in an *Alternative Representation* implemented by parsing the source code, or by abstract interpretation of the byte code. We now return to our motivating example, the task of listing the methods *required* by a class.

A method m can be required either because m is explicitly marked as such, or because a m is not implemented by a class or its superclasses, but m is sent to self in a method of that class. Because self sends can be conservatively estimated statically, m is definitely used but not implemented, and therefore its implementation is required to make the class complete.

How can we determine whether a method is required in a class? The simplest thing that could possibly work would be: for every selector s that may be required, scan every method implementation m 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*s, each built on a simpler one. This leads us to our next pattern: *Layered Extensions*, described below.

Layered Extensions

Context. You have a definition of an interesting and complex model extension.

Problem. How do you implement it efficiently, while at the same time promoting reusability?

Forces. Calculating this model extension requires as input other, expensive to compute information. The other computation might itself be useful as a property. The complicated property calculation 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* 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 requirement can be addressed by *Lazy Update*, described on page 29.

Consequences. When implementing *Layered Extensions*, 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. Note that this pattern is a specialization of the *Layers* [8] pattern.

□

In our work, we have also 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 our language for discussing the design 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 then 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 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* [15] 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 problems is that only a small part of the program is in view at any one time, and thus the extensions need only complete their analyses for that part — if only they could somehow know *which* part! *Explicit Interest*, described on page 23, provides a way for a view to notify the model that someone is interested in a particular part of their domain without the model having to care about *who* is interested.

The rest of this subsection presents patterns for dealing with real performance issues once they have become apparent. Performance improvements sometimes bring a cost in complexity, which may in turn adversely affect correctness. This is the topic of the following subsection (3.4).

When and how do performance issues arise in computing a *Model Extension* (p. 10)? 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 requires a much more sophisticated implementation, but supports new styles of work [33]. Thus, the frequency of use of an analysis can change from once every few minutes to multiple times per second.

The 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* [15] might be justified. Beck discusses *Caching*, and observes that the biggest problem with caches is ensuring that they remain valid; to limit the complexity of the task of maintaining validity of the cache, he suggests reducing the scope and extent of the variable used for the cache. Thus, a caching temporary variable is usually quite easy to maintain as valid; a caching instance variable less so [4, 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.

The third volume of *Pattern Oriented Software Architecture* [21] presents a set of patterns for managing resources efficiently. A *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. Going beyond *Caching*, *Lazy Acquisition* suggests that delaying the moment of requesting a resource may improve performance, because sometimes the resource will not be used after all. Applying this pattern to our domain, we might want to delay the computation of a model extension until its value is actually requested. *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 evicting a *Model Extension*?

As we mentioned in the introduction, one of the ways to solve this problem is to give the implementation of a *Model Extension* more information about the intentions of its clients. First we explain this idea using our example; we then present two patterns that describe a particular interface and implementation.

In our running example, the *Model Extension* pattern gives clients a simple interface for obtaining the requirements of a class. Unfortunately, this simple interface makes it hard for the implementation to achieve good performance. 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 client happens to request subclasses before their superclass, the algorithm would repeatedly calculate the same information. A client 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 it 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: the most efficient way to compute an *Inverse Mapping* is also to traverse all the relevant code once, gathering information about references to just those elements that clients will to ask about. In fact, this situation will be very common when using a non-local code analysis.

One way of giving the calculation mechanism this extra information would be to provide an additional "bulk" interface. For example, we could provide an additional interface for required methods so that the client tool could request 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, it would encourage premature optimization. Therefore we choose 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 *Bulk Calculation* (p. 28), *Minimal Calculation* (p. 27) and *Lazy Update* (p. 29) address related issues.

Explicit Interest

Context. You have a model extension with calculations that are significant enough to require optimization.

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

Forces. 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. Having more information about the usage of a property can mean that we are better able to

optimize the process of obtaining it. For example, caching all model extensions over the whole code model may be too expensive in space and invalidation costs, whereas caching these extensions over the much smaller part of the model that supports the programmer's current "working set" is feasible and cost-effective. However, focussing the caches in this way requires more coordination — cache usage patterns are dependent on the tools, but cache invalidation strategies are specific to model extensions. The code model is not completely static, because the user will occaisonally type in a change, or modify significant parts by loading a new version of parts of the code. Taken together, these forces mean that a wide variety of implementation tricks may need to be employed to make the different model extensions fast enough. Nevertheless, the interface to each model extension should still be simple and uniform.

Solution. Allow 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*.

Consequences. We may now 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 to assume that the space cost 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 the Observer pattern 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 only 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. AB ▶*Is there a pattern-literature description of reference counts that we should cite here?*◀

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. 23) 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[1]. The OmniBrowser is highly object-oriented: an OmniBrowser window does not display mere text strings, but model-specific *node* objects that correspond to (and reference) 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. A client creates a customized OmniBrowser by creating new node objects with custom behavior; the class of the node objects used in a specific browser is determined by a meta-model that thus defines the behavior of that browser. The OmniBrowser framework instantiates node objects as needed to support a user navigating through the code.

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 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 on the following page.

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

Life-long Interest

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

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 consist with the framework?

Forces. 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 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.

Related patterns and variants. *Suggestions welcome!*

□

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 recalculating 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.

We want to calculate requirements for as few classes as possible; *Explicit Interest* (p. 23) helps because we now know which classes have clients interested in them. Another important thing to know is what classes have changed since the previous calculation, and how these changes affect the requirements. Finally, we need to understand the scope of the requirements calculation: it turns out that the requirements of a class depend only on the definitions of the class and its superclasses. The scope of calculations is controlled by the pattern *Minimal Calculation* .

DV ►*Perhaps this should change to present a merger of the three patterns Lazy Update, Bulk Calculation and Minimal calculation as a calculation strategy that combines many of the ideas in this section in the context of Explicit Interests, for some difficult cases.* ◄ AB ►*I don't think so; I believe that it is useful for didactic reasons to present these patterns separately, even though they will often be used in combination.*◄

Minimal Calculation

Context. You have a model extension, and an interface through which clients express *Explicit Interest*.

Problem. How do you avoid unnecessary re-computation in response to changes in the *Shared Code Model*?

Forces. The code model is large, and changes slowly during ordinary editing and browsing operations. At any particular time, only a small part of the code model affects the user's display.

Solution. Use notification events on the code model to update only those model extensions that are *both* affected by changes in the code model *and* are interesting. If the update is inexpensive, it may be perfectly satisfactory to do it immediately after a change is notified. Otherwise, *Bulk Calculation* and *Lazy*

Update on the following page can help to determine how and when to perform the updates.

Consequences. Whilst it is true that the code model changes slowly during ordinary editing, when a new package or file is loaded into the development environment, many parts of the model change at the same time. Similarly, a global refactoring (such as re-organizing the class hierarchy) may cause a large number of changes to the model.

Related patterns and variants. Since interactive response is not required until the end of the package-loading or refactoring process, *Bulk Calculation*, described below, can be used to avoid slowing down these operations by repeatedly recalculating properties that will soon be invalidated.

□

Having minimized the set of classes for which we recalculate the requirements, we are now concerned with doing the calculation as efficiently as possible. Requirements have non-local aspects. For example, an unimplemented method selector may be self-sent by some superclass, making the corresponding method required. To find this we need to check the code of all superclasses. This is reflected in Schärli's algorithm, which first updates a class and then its subclasses. These non-local aspects of the requirements calculation make it attractive to update the model extension in a *Bulk Calculation*, rather than one class at a time.

Bulk Calculation

Context. You have defined a *Model Extension* (p. 10) that depends on non-local aspects of the *Shared Code Model* (p. 8), for example, the value of a property for a class depends on properties of all of its superclasses. The interface to the model extension allows clients to express *Explicit Interest* (p. 23) in parts of the model.

Problem. You need to avoid repeated re-calculation of the model extension.

Forces. The tool supported by the model extension might request the the extension's properties for multiple classes in succession. Naïvely satisfying these requests would result in multiple traversals of the common parts of the model on which they depend, which would be prohibitively expensive.

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 is calculated for all interesting code elements, at the same time.

Consequences. *Bulk 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.

Related patterns and variants. The timing and ordering of this re-calculation may be determined by *Lazy Update* .

□

Observe that the problem of displaying code properties maintained by a model extensions 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. *Bulk Calculation* uses *Explicit Interest* (p. 23) to know in advance what properties it will be requested; it also uses the code change notification mechanism of the *Shared Code Model* (p. 8) to batch the required work. To avoid doing unnecessary work, we wish to perform the bulk calculation as late as possible. This is supported by *Lazy Update*, described below.

Lazy Update

Context. You have defined *Layered Extensions* (p. 20) on top of a *Shared Code Model* (p. 8).

Problem. How do you determines when, and in what order, the properties that underlie the model extensions should be recalculated?

Forces. 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 be efficient but need not be interactive. The order in which the model extensions are updated should be consistent with the inter-layer dependencies of the *Layered Extensions*.

Solution. Update the model extensions lazily, that is, only in response to client queries. Laziness has several benefits.

1. Model extensions that are not needed are not calculated.

2. Bulk changes will be executed without useless intermediate updates to the model extensions. The eventual update will be efficient due to *Bulk Calculation* (p. 28).

3. Recalculation of *Layered Extensions* is performed in an appropriate order.

However, this pattern also has some disadvantages. The first access to a model extension after an update will be less responsive than other accesses. When used in combination with *Bulk Calculation*, this pattern is not completely lazy, so specific parts of a model extension may be calculated in spite of not being requested. Perhaps the most significant disadvantage, in comparison with eager evaluation, is that Lazy Update does not support change notifications on the model extensions.

Consequences.

Related patterns and variants.

□

For the *requirements* calculation, we applied all three of the above implementation patterns. *Bulk Calculation* allowed us to use Schärli's algorithm as it was designed to be used, and *Lazy Update* allowed bulk changes to the code to be made efficiently.

Lazy Update also ordered the updating of the *requirements* with the *self senders of* extension that it uses. In the implementation of *self senders of* we used a trivial combination of these patterns — the cache for a class is invalidated when the class is modified, and recalculated when it is requested. The recomputation of *self senders of* is restricted 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, by definition, local for each class.

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 on the following page.

Canonical Implementation

Context. You have implemented a useful model extension. Its definition is not trivial, and the simplest implementation is not fast enough.

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

Forces. The calculation of a model extension must be fast enough for interactive use. This necessitates optimizations that make the code complex and harder to verify and trust. 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 optizations, 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 used by client tools. Write tests to 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 is independent of the assumptions in the efficient implementation..

There are a number of reasons that separating the two implementations allows you to create and maintain a correct and understandable *Canonical Implementation*.

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 the interface to the canonical implementation, so it can correspond more closely to a *Formal Definition* (p. 33).

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.

Realistically, the need for this pattern will not becomes 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.

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 grew convinced that by taking into account which classes had been modified, and which classes implemented or self-sent each method selector, we could run the *requirements* algorithm less frequently. 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 only be possible if we had a formal definition of the *requirements* property. Some relevant formal definitions already existed [11] but were not particularly well-adapted 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 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 on the next page.

Formal Definition

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

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. How can you convince yourself that they are correct?

Forces. The programming language that your environment 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 at dealing with 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.

Solution. Use mathematical language to define the property formally, in terms of primitive facts about the programming language, and simpler properties. When an optimization relies on a non-trivial claim about the property, prove the claim from the formal definition. 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.

Consequences.

Related patterns and variants.

□

4 Evaluation

A pattern language is useful if it leads to an improvement in the architecture, functionality, performance or reusability of software that adopts it. We will evaluate model extensions by this criterion in two cases: the implementation of the requirements browser that motivated the development of the patterns, and a proposed refactoring of the relevant parts of the Chuck type inference system.

4.1 The Requirements Browser

The original version of the Requirements Browser [34] was implemented as part of an incremental programming environment. One of the principal goals of this environment is that it show the programmer the actual state of the code being developed in real time; this includes what methods are still missing, and which classes are incomplete. Meeting this goal requires responsiveness during typical browsing activities, and the constant display of requirements information. These are difficult requirements to satisfy, because the requirements calculation is non-local, and potentially quite expensive.

In the initial prototype, all of the self-, super- and class-send information for every method in the image was calculated eagerly and cached in a compressed form, at a significant space cost. This cache was subsequently replaced by a custom abstract interpreter that computed the send information on demand from the byte codes. However, achieving responsiveness still required that the *Inverse Mapping* 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 show 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, took tens of minutes.

The patterns proposed in this paper 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.

Our pattern language also improved modularity. The original Requirements Browser prototype added the implementation of the *required* property directly to the *Shared Code Model*; the use of a *Model Extension* allow us to avoid this modification of core system classes.

4.2 The *Chuck* type inferencer

Chuck is a type inference system for Smalltalk, implemented in Squeak [37, 38]. Scalable type inference for dynamically typed languages such as Smalltalk is a

lively research topic. One of the difficulties with inference tools is performance; achieving high performance is complicated by the need for global analyses of control and data flow. In particular, this means that Chuck creates and maintains several eagerly updated caches of computed properties of all code in the system:

1. The parse trees for methods.
2. What expressions in the code send specific messages.
3. Which classes implement each message.
4. Which expressions assign values to each variable.
5. Which expressions read each variable.

Each of these properties could instead be implemented as a *Model Extension*. These would be *Layered Extensions*, with parse trees used as a base layer supporting several of the other properties. According to internal documentation in the Chuck demo, these caches take tens of megabytes — a significant cost that is borne even if no tool using type inference is active. In this case, using the *Explicit Interest* pattern would reduce the amount of memory used by these caches, as well as reducing the cost of keeping them up to date. Whether the implementation patterns described in this article would provide significant *performance* benefits when Chuck is actively used depends on the cache access patterns of the type analysis algorithm; pending re-implementation of Chuck using these ideas, we make no claims. However, several of the *architectural* benefits we claim would clearly accrue in this case. Each of the layered model extensions would be valuable to other tools. As model extensions, they would all benefit from a single global cache, rather than each having to implement and store its own cache. For example, the SmallLint [7] tool would benefit from the parse tree cache, and the Refactoring Browser [31] would benefit from all the others.

From a more detached perspective, Chuck can be thought of as consisting of two components. The first is an engine for computing difficult-to-obtain type information about code elements. The second is a tool that uses this type information to provide answers to some explicit queries useful for active program understanding. However, type information is useful also for more frequent, less explicit activities. For example, in IDEs for statically typed languages, types are very commonly used for high-precision auto-completion [28]. In Smalltalk, precision auto-completion is more difficult because types are not given explicitly in the code, and inferred types are usually partial; if Chuck's inference engine were easily available to other tools, it would be a simple matter to use it to build an accurate auto-completion tool. In fact, the *eCompletions* package [2] does currently provide exactly this service, but it does so by using type information from a completely separate inference engine called RoelTyper [42]. This illustrates that programming tools would indeed benefit from inferred types being conveniently exposed. Since type information is naturally tied to specific elements of the code model (in this case, variables and other expressions), is expensive to compute, and is invalidated by updates to the code, we consider

it likely that significant benefits can be gained by treating type as a model extension and applying the patterns proposed in this paper.

In both of these examples, applying our pattern language provides a way to make existing, valuable, analyses more widely reusable, while solving various architectural problems that are common to the domain of program manipulation tools. We note that while the particular examples that we chose — abstract classes and types — are both 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.

5 Related work

The idea of multiple-view software development environments has been studied at least since 1986 [16], when David Garlan began the work that led to his doctoral thesis [17]. 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 [27] examined novice users of Field and concluded that multiple views, or at least the particular views that Field supported, did indeed help programmers 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" [25]). 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 paper 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 [23] presents an architecture (also called MultiView) 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 [40], and also make it easier for the programmer to transform the code by permitting it to pass through syntactically illegal intermediate states.

Meyers and Reiss [26] 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 paper 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 multiple representations are made available (as model extensions) to help support disparate views. Thus, the research into advanced representations such as SPGs can be leveraged by using these representations as model extensions.

The overarching goal of the Multiview project is in many ways similar to that of Charles Simonyi's "Intentional Programming": the development of a model for programs that is far richer than text, and which seeks to make explicit the intention of the programmer who wrote the code. In the current Intentional Programming system, under development at Intentional Software, the program — and all information about the program — is represented in a uniform tree-like form, and stored in a versioned database. The details are proprietary. [36]

Riehle and Züllighoven [29] describe a pattern language for the construction of software systems based on the Tools and Materials metaphor. Their language is also concerned with defining the interface between tools and the domain objects so that the environment is extensible, but addresses different aspects of the problem from our work, and makes some different assumptions. For example we strive for the ability to add properties to a fixed set of code elements, while they consider the properties fixed and make the system extensible with new kinds of materials that enjoy those properties.

The Cadillac programming environment [13], 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 this 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. We do not know if other patterns discussed in this paper were used in Cadillac.

One of the main concerns of the patterns that we present in this paper is

that of supporting efficient response to queries without complicated interfaces. Automated incrementalization, as proposed by Liu *et al.* [22], also addresses this problem. They note that maintaining the consistency of cached information over updates is a aspect of the implementation that needs to be spread across all the update operations. They propose to achieve this by automatic code transformations using programmer-specified cost information and rewrite rules. Such an approach may at some point free the programmer from some of the more mundane optimizations that we mention.

The Intensional View Environment (IntensiVE [24]) uses SOUL to specify subsets of the program elements, thus helping the programmer to maintain invariants about the program structure. SOUL is used in different ways in other papers, but in this context we consider it to be an example of a domain-specific language (DSL) for the specification of model extensions. The approach of specifying model extensions using a DSL has the benefit that general calculation and caching strategies can be deployed once and for all in the implementation of the DSL.

Eclipse is a very well known, quite widely used development environment focused on extensibility. Since it is openly developed, extensible and popular, it is interesting to consider whether the patterns that we have proposed are relevant to its architecture. To this end we consider Eclipse as an extensible development environment for a specific language and limit our comments to the part of Eclipse that implements an interactive development environment for Java: the JDT. Our remarks are based on the published APIs [12] and documentation [28] for extenders.

The JDT includes a Java model in which objects represent elements of the program under development. It provides a way to read and modify code, and includes a change notification mechanism. Because Java does not support class extension, Eclipse makes extensive use of Gamma's Extension Object pattern [14]. Many elements in the Java model implement the org.eclipse.core.runtime.IAdaptable interface; this interface consists of the single method getAdapter(Class) and returns an object of requested class, or null if none can be found. getAdapter plays the role of Gamma's getExtension operation. A centralized registry keeps track of which classes of objects can be adapted to which other classes.

This mechanism can be used as a way to implement a *Model Extension*. Since in Eclipse version 2.1 the Java model was not kept in memory, the model was inappropriate for queries that required traversing the code of a whole project. However, in Eclipse version 3.1 the Java model *is* kept in memory, and thus fits the *Shared Code Model* pattern. It therefore seems that it should be possible to apply the other patterns proposed here to extend Eclipse with code analysis tools, although we have not yet studied in any detail how feasible this will be, or what modifications to our patterns will have to be made apart from the use of the Extension Object pattern instead of class extensions. We venture one comment on the implementation of our patterns in this context. Eclipse fosters extensibility by defining standard interfaces that plugins of different kinds must

implement. If it is useful in Eclipse to use *Model Extension* and *Explicit Interest* to expose code analyses, then the interfaces for expressing interests should also be defined in a standard way, so that *Layered Extensions* can be contributed by different parties.

6 Discussion and future work

It used to be that conducting research in program development tools required either settling for a mediocre user interface (making it unlikely that the experimental 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 provide a context in which the investment required to move from the idea for an analysis to a usable tool is much lower. Because these environments are widely used, it is also more likely that a tool that works in such an environment will be adopted, compared to a standalone tool.

The patterns presented in this paper are intended to continue 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* as a framework within which code analyses can be added systematically, so that these analyses are shared between different extenders of the development environment, and so that one analysis can build on the results of another.

While some tools use shared code models, we have not yet found other examples of *Model Extension*s as proposed here. Thus the architecture we propose, while implemented and found useful by us, has not yet been found "in the wild". It seems likely that some of the proposed patterns will change, and possibly more will be added, as experience with this architecture increases. For example, we do not yet know what the best interface is for expressing *Explicit Interest*.

The model extensions that we propose all create new properties of existing elements in the *Shared Code Model*. However many of the code analyses in the literature result in the creation of new entities that are not local to any such element. One example of this is finding cyclicly dependent sets of classes [39]. The evaluation strategy that we propose for model extensions in this paper is *Lazy Update*. Eager evaluation has the potential benefit of allowing a model extension to be *observable* in the Observer pattern. It is not clear how to resolve the tension between the usefulness of the Observer pattern and the performance costs of eager update.

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. Our implementation is available as a Squeak image at http://www.cs.pdx.edu/~black/goodies/. It has begun to solve the very real problem first identified by Meyers in 1991 [25], 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 report their findings.

7 Pattern Language Summary

We close with a quick overview of our solution to the problem of building a extensible, modular architecture for representing a program. We indicate with a *slanted sans-serif font* the names of the patterns that are described in detail in the body of the paper.

One important property of the Smalltalk programming environment is that it has a *Shared Code Model* (p. 8) 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. 10) 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. 16) 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. 23) 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. 26), 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. 27) and *Bulk Calculation* (p. 28). *Lazy Update* (p. 29) 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. 33) and a *Canonical Implementation* (p. 31) as a test oracle. We found that the (rather complicated) requirements property depends on two simpler properties, which led us to *Layered Extensions* (p. 20). 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 a member of a more general class of *Inverse Mapping* (p. 18)s, which we hypothesize are frequently useful both to the end user and to the builder of more complex extensions.

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. 14) 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.

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, 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, and to our PLOP2006 shepherd Peter Sommerlad for his extensive advise and many useful comments..

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] Ruben Bakker. eCompletion for squeak, November 2004. http://homepage.mac.com/monique_bakker/squeak/eCompletion.html.

[3] Kent Beck. Simple Smalltalk testing: With patterns. http://www.xprogramming.com/testfram.htm.

[4] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[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] John Brant, March 2006. http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html.

[8] 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.

[9] 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.

[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] The Eclipse Foundation. Eclipse 3.1 documentation, 2005. http://help.eclipse.org/help31/index.jsp.

[13] 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.

[14] Erich Gamma. Extension object. In *Pattern languages of program design 3*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[16] 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.

[17] David Barnard Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, January 1988.

[18] 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.

[19] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2001.

[20] 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.

[21] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture Volume 3 – Patterns for Resource Management*. John Wiley and Sons, 2004.

[22] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 473–486, New York, NY, USA, 2005. ACM Press.

[23] 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.

[24] Kim Mens, Andy Kellens, Frederic Pluquet, and Roel Wuyts. Co-evolving code and design using intensional views — a case study. *Journal on Computer Languages, Systems & Structures (to appear) (pre-print downloadable)*, 2006.

[25] Scott Meyers. Difficulties in integrating multiview development systems. *IEEE Softw.*, 8(1):49–57, 1991.

[26] 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.

[27] 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.

[28] Object Technology International, Inc. Eclipse platform technical overview, 2003. White paper.

[29] 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.

[30] Linda Rising, editor. *Design Patterns in Communications Software*. Cambridge University Press, 2001.

[31] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[32] Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, February 2005.

[33] 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.

[34] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 30:79–95, 2004.

[35] 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.

[36] Charles Simonyi. Interview with Code Generation Network, July 2004. http://www.codegeneration.net/tiki-read_article.php?articleId=61, accessed May 2006.

[37] Lex Spoon. Chuck: Type inference for program understanding, March 2006. http://www.lexspoon.org/ti/.

[38] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In Martin Odersky, editor, *Proceedings of the* 18^{th} *European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 51–74, Oslo, Norway, June 2004. Springer.

[39] Daniel Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[40] Michael L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, October 2002.

[41] Roel Wuyts. Star Browser. http://www.iam.unibe.ch/∼wuyts/StarBrowser/.

[42] Roel Wuyts. Roeltyper, November 2004. http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/.