A Browser for Incremental Programming

Nathanael Schärli [a,*] Andrew P. Black [b]

[a] *Software Composition Group, University of Bern, Switzerland*

[b] *OGI School of Science & Engineering, Oregon Health & Science University, USA*

Abstract

Much of the elegance and power of Smalltalk comes from its programming environment and tools. First introduced more than 20 years ago, the Smalltalk browser enables programmers to "home in" on particular methods using a hierarchy of manually-defined classifications. By its nature, this classification scheme says a lot about the *desired* state of the code, but little about the *actual* state of the code as it is being developed. We have extended the Smalltalk browser with dynamically computed *virtual categories* that dramatically improve the browser's support for incremental programming. We illustrate these improvements by example, and describe the algorithms used to compute the virtual categories efficiently.

Key words: Smalltalk browser, incremental programming, intentional programming, method reachability, requires set.

1 Introduction

The most important of the Smalltalk programming tools is the *Browser*, which allows the programmer to examine, modify and extend the code of applications and of the system itself. The Browser was revolutionary when it was first introduced, and over the intervening years it has been improved in several ways. For example, semi-automated refactoring has been added, leading to a tool known as the Refactoring Browser [1], and in many Smalltalk dialects some form of package construct has subsumed the original primitive categorization of classes. Nevertheless, the way in which today's browser organizes the methods of a class is essentially the same as in 1980: a hierarchy of manually assigned "protocols".

In the intervening years, the concept of the "Integrated Development Environment" — for that is what the Smalltalk toolset would now be called — has proved

* Corresponding Author

Email addresses: `schaerli@iam.unibe.ch` (Nathanael Schärli), `black@cse.ogi.edu` (Andrew P. Black).

to be so successful that similar environments have been created for other programming languages. For example, IBM's VisualAge for Java [2] was essentially a retargeting of the Smalltalk IDE to Java; more recently the cross-language environment Eclipse [3] has made similar tools available for many popular languages. As these IDEs have evolved, they have started to perform on-the-fly analysis of the code, enabling a typical modern IDE to provide the programmer with real-time feedback such as syntax coloring and message prompting.

The raw material for real-time feedback is adequate computational power. Today's Smalltalks run on a commodity laptop about a thousand times faster than they ran on commodity machines of the early 1980s, and about fifty times faster than on the custom hardware of the Dorado. This power has been harnessed to write Smalltalk applications for real-time music and motion picture manipulation that, twenty years ago, would have been unthinkable. But the power has *not* been used to keep the programmer informed of the changing relationships between the entities that make up an evolving program.

In this paper, we present a tool that uses a real-time analysis of the program being modified to do exactly this. The analysis enables the tool to infer and display system-wide information about the structure of the program, thus actively supporting incremental programming and program understanding.

The basic idea is to place the methods of a class into "virtual categories", *i.e.*, categories that are computed automatically, that are instantly available for display, and that are always up to date. The *requires* category of a class C contains all the messages that are sent to **self** by the methods of C and its superclasses, but for which methods are neither defined in C nor inherited. We also compute the *supplies* category, which contains the concrete methods of C that implement inherited abstract methods, and the *overrides* category, which contains the concrete methods of C that override inherited concrete methods. The last virtual category, *sending super*, contains the methods that send messages to the pseudo-variable **super**.

Of these virtual categories, the first — the set of required methods — is the one that we have found to be most useful. The availability of this category supports "programming by intention" [4], a top-down style that encourages the programmer to think about *what* has to be done rather than about *how* to do it. The idea behind programming by intention is to imagine methods that do "the hard part" of one's task, so that all that one has to do to complete the task is send the corresponding messages. Of course, later one applies the same idea to defining the "hard" methods. In this paradigm, the *requires* category provides a constant reminder of what is left for the programmer to do.

We have implemented all of these virtual categories in a browser for Squeak, an open-source dialect of Smalltalk [5,6]. In addition to being the most useful, the *requires* category also turns out to be the trickiest to define and implement effi-

ciently; to the best of our knowledge there is no other browser that displays it. In contrast, the definition and computation of the other virtual categories is mostly straightforward, and recent versions of other Smalltalks (*e.g.*, VisualWorks [7] and Squeak [6]) also compute some similar properties and use the results to decorate method names in the browser. However, no other browsers use this information to group methods into virtual categories that supplement the programmer-defined categories and provide the programmer with multiple views on the code. A Squeak image is available that demonstrates our browser and contains its source code [8].

This paper makes three contributions. First, we analyze the information that a programmer needs in order to be effective at incremental programming (section 2). Second, we describe the way that our browser supplies this information, and report on our experience using the extended browser and the way in which it changes the programming process (section 3). Third, we define the *required* set and describe an algorithm to compute it that runs in real-time (section 4).

2 Incremental Programming and its Demands

Even though incremental programming has been part of the Smalltalk ethic from the earliest days, it seems not to have been singled-out as an important feature of the language. Ingalls' seminal paper on the design principles behind Smalltalk [9], for example, does not mention support for incremental programming. So it is necessary for us to ask ourselves what incremental programming really is, and how a development environment should support it.

Fortunately, we do not have to look far for our answers. Over the last few years, Kent Beck and others have codified a set of practices called Extreme Programming (XP) [10], which capture very clearly the essence of incremental work: not just incremental programming, but also incremental requirements gathering, incremental design, incremental planning, and incremental deployment. This means that instead of bringing the work on each of these activities to an end before proceeding to the next, programmers iterate through all of the activities multiple times, each time doing only as much work as is necessary to achieve the current (incremental) goal. We will take the "extreme" style of programming as a paradigmatic example of what a good incremental programming environment should support.

2.1 Programming in an Extreme Environment

In an "XP shop" the incremental and iterative style of work is applied not only to the major development activities, but also to the work within each activity. During implementation, this means that classes are not written sequentially, one after the other, and finally executed only when they are all complete. Instead, a programmer

works on several classes at a time, and combines and tests them as soon as a fraction of the functionality is implemented.

The following patterns of work seem to be central to incremental programming.

(1) *Programming with limited knowledge.* A programmer starts implementing a part before he has full knowledge of the whole system.
(2) *Working in multiple contexts.* A programmer may be working on several classes in parallel. For example, he may start implementing a new method before finishing the implementation of another. As a consequence, he often switches working context.
(3) *Understanding how classes collaborate.* When working with multiple incomplete classes at once, the programmer needs to understand how these classes work together and how changes in one class affect related classes.
(4) *Understanding what is still missing.* With incremental programming, it is important to know what parts are still missing in order to make a program, or at least a part of it, complete. The missing parts are a good indication of what the programmer should work on next.
(5) *Refactoring.* A consequence of starting to program with limited knowledge is that it is likely that the implementation will have to change as the programmer becomes more knowledgeable. XP says that we should embrace change, not be fearful of it. Refactoring is the secret weapon that enables us to combat the increasing entropy and code rot that would otherwise be the result of continual redesign and re-implementation.
(6) *Testing.* Writing and maintaining tests is a corner-stone of incremental and iterative programming. Tests are used to capture the intended behavior of a feature as soon as that feature is implemented—or even before. Tests make it possible to refactor quickly and safely, and help us to understand what is still missing.

2.2 Supporting Incremental Programming in Smalltalk

Smalltalk's integrated environment is designed to support experimental programming, and encourages the programmer to intermix design, coding, testing and debugging [11]. Thus, it is not surprising that the language and environment support four of the six patterns of work identified above (1, 2, 5 and 6).

Since the Smalltalk language is not explicitly typed, the programmer does not have to specify the types of method arguments or instance variables when they are introduced. As a consequence, programming in Smalltalk requires less knowledge about the design of the whole system in order to start implementing (1).

The earliest implementations of Smalltalk-80 came with a windowing system that allowed one to open multiple browser windows and work concurrently in multiple contexts (*e.g.*, with multiple classes and methods) (2). More recently, this capability

has been significantly improved with new browsers such as Whisker and the Star Browser, both of which allow one to work in multiple contexts without having to open multiple windows. (See section 5 for more information about these developments.)

Since the introduction of the Refactoring Browser, Smalltalk has been able to command a rich set of tools for semi-automated refactoring (5). Incremental testing was originally supported by the practice of writing executable comments and example methods. Because of incremental compilation [11], and because no checks for completeness are performed at compile time, it is possible to test a method immediately after it is written, even though other methods on which it depends may be incomplete or absent. Since the introduction of the SUnit testing environment, it is now also possible to effectively organize these tests (6).

Unfortunately, Smalltalk provides only limited support for the two remaining patterns of work, which means that it is relatively hard to understand how different classes relate to each other (3) and what is still missing to make them complete (4).

We believe that these problems can be solved by providing the programmer with statically accessible and always up-to-date information about ways that the different classes of a system collaborate. The task of providing this information can be broken into several subtasks. First, we need to identify what kind of information is needed to understand how the classes collaborate. Second, we need to be able to compute this information in real-time. Finally, we need to find a way to make this information readily accessible to the programmer.

In the rest of this paper, we explain how we completed these tasks for the most common form of collaboration between classes: inheritance. In section 5, we discuss how the same techniques can be applied to help the programmer understand aggregation and delegation, the other major forms of collaboration in object-oriented languages.

3 The Browser

The previous section identified some limitations in the support offered for incremental programming by existing Smalltalk browsers. Now we describe our new browser and how it overcomes these limitations. First, we discuss the categories of information presented by our browser, and explain why this information makes it easy to understand inheritance collaborations. Then we give an "illustrated walkthrough" of a programming session, showing how the browser facilitates incremental programming and helps programmers to understand existing class hierarchies.

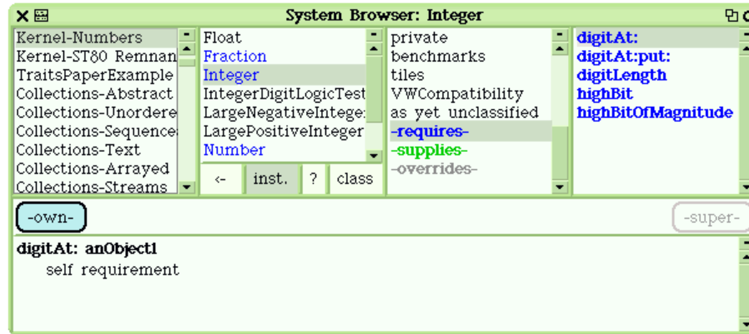

Fig. 1. The browser examining the class Integer.

3.1 Information Presented by the Browser

Our browser is shown in figure 1. At first glance it looks like the standard Smalltalk browser, but a few extra features are visible. In the figure, the class Integer is selected in the class pane (the second from the left). The third pane, which in the standard browser contains a manual categorization of the methods of the selected class, now contains in addition some *virtual categories*. These categories present information that we identified as key to understanding how a class collaborates with its neighbours in the inheritance hierarchy.

The category *-requires-*, which is colored blue, includes all of the methods that the class Integer sends to itself but does not define or inherit. In figure 1, this list of methods appears in the fourth pane, where we have selected digitAt:, which is consequently displayed in the code pane at the bottom of the browser. The implementation shown, self requirement, is a marker method generated by the browser to indicate that digitAt: is an unsatisfied requirement. The same holds for the requirements digitAt:put: and digitLength. The remaining two requirements, highBit and highBitOfMagnitude, do in fact have implementations: self subclassResponsibility. [1] However, the browser recognizes this as a marker method and still categorizes these methods as required. The requires category tells the programmer how the class Integer is parameterized, and which methods are necessary in order to make it complete. [2]

The next category, *-supplies-*, lists methods that are required by some other class and provided by the class that we are browsing. In the case of Integer, this virtual category contains 10 methods including *, +, −, /, <, =, and hash. This tells the programmer that Integer's superclass (Number) is parameterized by these 10 methods and shows the concrete implementations that Integer supplies for these parameters.

The third category, *-overrides-*, lists those methods provided by Integer that over-

[1] A method with body self subclassResponsibility is the standard way of indicating that a method is abstract, *i.e.*, that it is the responsibility of a subclass to provide it.
[2] Note that in Smalltalk, Integer is the abstract superclass of the concrete classes SmallInteger, LargePositiveInteger, and LargeNegativeInteger.

ride inherited methods. In our example, there are 11 overriding methods, including //, asInteger, even, floor, and isInteger. The *-overrides-* category is important for two reasons. First, it provides a view of the class as a "delta"; the methods in this category characterize the parts of the behavior of the superclass that are changed by this subclass. Second, the overriding methods, together with the supplied methods, are the most critical for understanding the class Integer and reasoning about its correctness. This is because inheritance breaks encapsulation [12]: subclasses collaborate with their superclasses through a much broader interface than the public interface of the superclass. Taken together, overridden and supplied methods represent the hooks through which Number collaborates with Integer. In particular, this means that the behavior implemented in each of these methods of Integer needs to conform to the specification implied by Number, and that these are the methods in Integer that must be adapted if the specification implied by Number changes.

The fourth virtual category is called *-sending super-*; it contains the methods that perform *super sends* (*i.e.*, message sends that cause the message lookup to be started in the superclass). This category is important because it tells the programmer which of the methods collaborate with behavior from their parent class. Furthermore, all of these methods depend explicitly on their position in the inheritance hierarchy, so special care has to be taken if they are moved during refactoring. This category is not shown in the figure because it is empty. In fact, all the virtual categories are displayed by the browser only when they are not empty.

Although it may not be clear from the grayscale figures used by this journal, each of the generated virtual categories has a characteristic emphasis: blue for *requires*, green for *supplies*, grey for *overrides*, and underlined for *sending super*. Even when browsing methods using the ordinary, manually-defined method categories, the names keep their characteristic emphasis. So, a supplied method that sends to super will always show up in green and underlined. The blue color-coding is also applied to the name of the class itself in the second pane whenever the set of required methods is not empty. This serves as a reminder that the class is incomplete: it may be an abstract class such as Integer, or the programmer may still be working on it.

3.2 Programming with the Browser

Now that the reader understands the kind of information presented by our browser, we will illustrate how this information facilitates programming in general, and incremental programming in particular.

As a working example, we will assume that we are about to implement a hierarchy of classes for collections in Smalltalk. We start by implementing the abstract class Collection, which serves as the root of our hierarchy. At this level, we will implement the most common interface methods such as isEmpty, includes:, select:, add:,

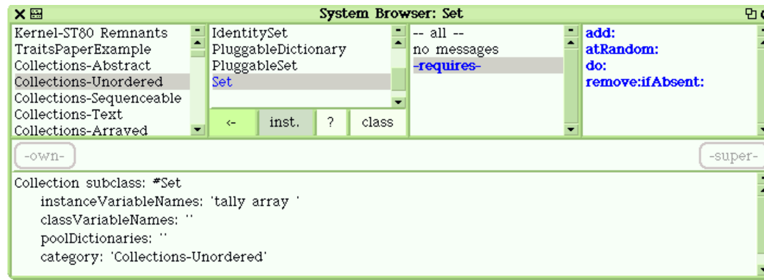

Fig. 2. The browser on the class Set just after it was created.

remove:, *etc.*. While writing these implementations, we proceed in an "intentional" style: we need not worry about sending messages that are not yet implemented. This is because our browser updates all the virtual categories in real-time, and as a consequence every message that is sent to self but not yet implemented immediately appears in the *-requires-* category.

As a concrete example, consider the method isEmpty, implemented as

```
isEmpty
    ↑ self size = 0
```

As soon as this method is accepted, a requirement size is created. It will not disappear until either it is satisfied (*i.e.*, a size method is implemented) or all the methods that require size are removed. The same thing happens when we implement the method select:, which requires the methods do: and emptyCopyOfSameSize.

The constant availability of the *requires* category gives us the freedom to implement the methods in the order that they come to mind without having to worry about forgetting to implement the methods on which they depend. Furthermore, many errors, especially typographical errors, can be detected immediately because they cause weird requirements to appear. A new menu item, *local senders of*... helps us discover why a method is required: it lists the methods in the current class hierarchy that send the selected message.

Once we have implemented the most essential Collection methods, we create the concrete subclass Set so that we can test our abstract implementation. Figure 2 shows the class Set just after it has been created as a subclass of Collection. At a glance we see that this class is incomplete (the name Set is blue), that it does not yet implement any methods, and that it requires methods for add:, atRandom:, do:, and remove:ifAbsent: to make it complete. (Although Set does not yet have any of its own methods, it inherits methods from Collection. These requirements are inferred by analyzing the inherited methods.)

In order to properly satisfy these requirements, we will eventually need to implement some internal methods that map the contents of the set to the instance variables array and tally. Again, the automatically computed list of requirements allows us to implement these methods in whatever order we prefer, while keeping a high-

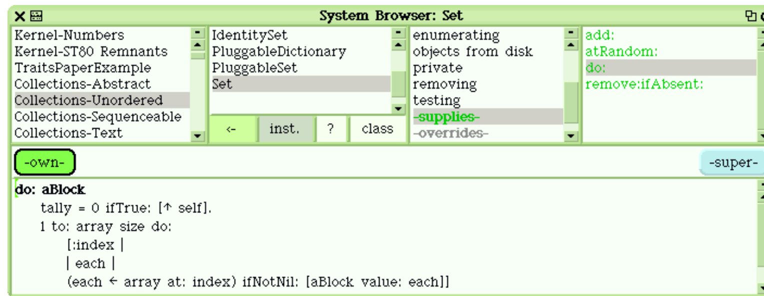

Fig. 3. The browser examining the class Set after it is completed.

level view of what is already implemented and what still needs to be done.

Figure 3 shows our browser on the class Set after it has been completed. Because there are no longer any unsatisfied requirements, the *-requires-* category has disappeared. Instead, the category *-supplies-* appears, showing how the implicit parameters of Collection are satisfied by Set. We also see the category *-overrides-*, which contains methods such as =, asSet, copy, includes: and size. These are the methods that the class Set overrides in order to modify the inherited behavior or to provide a more efficient implementation.

Since these two virtual categories capture much of the relationship of Set with its parent, they help the programmer to avoid "inter-level" errors between these classes. In other words, the methods in these categories are exactly the ones that characterize the collaboration between Collection and Set, and so they are exactly the ones that must be examined if the specification in Collection is changed. For example, if we change the way that Collections are compared, we need to ensure that the overridden method = is updated appropriately.

Let us suppose that, after testing Set, we move on to create some sequenced collection classes such as OrderedCollection and Heap. Since these classes will have some code in common, we decide to make them both subclasses of a new abstract class SequenceableCollection, which will itself be a subclass of Collection. When looking at these three classes with the browser, we see that they all require the same methods, namely add:, atRandom:, do:, and remove:ifAbsent:.

Instead of implementing these three classes one after the other, we can now take advantage of the freedom granted by incremental programming and work on all three classes in incremental steps. This means that we go through these requirements and the necessary internal methods and decide in which class each method should be implemented. During this process, which may also include refactoring actions such as pushing methods up or pulling them down in the hierarchy, the features of our browser always give us up-to-date information about what methods are required in each of these classes and how they collaborate.

To make this concrete, suppose that we start with the requirement do:, and decide that we should implement this in the class SequenceableCollection as follows.

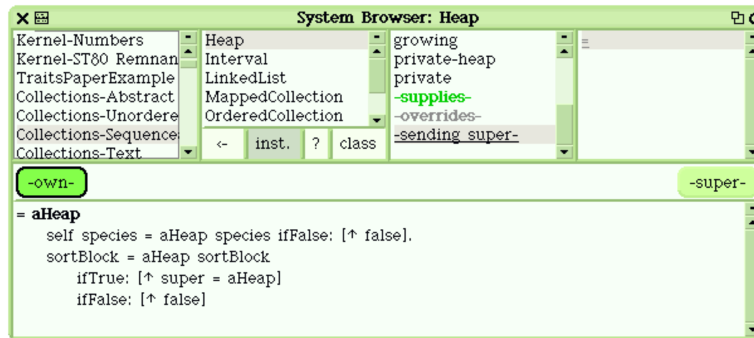

Fig. 4. The browser examining the class Heap.

```
do: aBlock
    1 to: self size do: [:index | aBlock value: (self at: index)]
```

As soon as we accept this code, the browser shows us that we have a new requirement at: in SequenceableCollection and its two subclasses. Since at: must be implemented differently in OrderedCollection and Heap, we proceed to implement the two at: methods in the subclasses. This is reflected in the browser, which shows at a glance that at: is now a parameter of the abstract class SequenceableCollection and that it is *supplied* by different methods in the two concrete subclasses.

Figure 4 shows the class Heap as it results from this process. We have selected the virtual category *-sending super-*, which contains a single method =. This tells us that there is only one method that uses the keyword **super** to access an overridden implementation in the parent class and which is therefore vulnerable to refactorings. The two other virtual categories, *-supplies-* and *-overrides-*, tell us how Heap implements the parameters of its abstract parent class SequenceableCollection, and which of the inherited methods it overrides.

3.3 Understanding and Modifying Existing Classes

In the previous section, we have shown how our extended browser helps one to create a new class hierarchy. In addition, our experience has shown that the browser is also a great help when one must understand and modify existing classes and class hierarchies.

As an example, consider the collection hierarchy that we have just created. The browser's virtual categories provide valuable guidance to a new programmer in understanding both the higher-level purpose and the lower-level implementation of each of the classes. This is because these categories separate the information that is essential to understanding the collaboration from the rest of the code, and thus make the overall architecture more explicit. Once the programmer has a basic understanding of the parent class, she will need to look only at a small part of the code of a subclass in order to understand how that subclass relates to its superclass.

These features are even more important when it comes to extending a class in an existing hierarchy with some new features. Such an extension is quite common, particularly if the hierarchy was created incrementally, or by following the XP doctrine of doing the simplest thing that could possibly work to satisfy the requirements that are currently available [10, Chapter 17]. Especially in complex hierarchies, it is quite easy to introduce "inter-level" errors. For example, the programmer might modify or extend a class and forget to make all of its subclasses compatible with these changes. Not only does the browser help one avoid making such inter-level errors, it also shows us which subclasses accidentally become incomplete as a result of the changes to the superclass.

Although it might seem that these problems are easy to avoid without virtual categories, our experience has shown that this is not the case. When we introduced our new browser into Squeak, we immediately found dozens of abstract classes that should actually be concrete, as well as hierarchies that exhibited other inter-level errors such as sending unimplemented super messages. The most surprising thing was that we found these errors even in the core of the system, which has been used by thousands of users for many years. For example, classes like Fraction, Bitmap, CharacterSet, Debugger, and nearly all subclasses of the class Morph—which is the root class of Squeak's user interface framework—are accidentally abstract, simply because programmers forgot to implement some methods.

We find this to be convincing evidence not only that our browser extensions are necessary to avoid such errors in the future, but also that they provide great help in finding and eliminating these errors in existing class hierarchies. Since abstract classes are "blue", finding them is trivial. Having the requirements continuously available in an up-to-date virtual category also makes it straightforward to identify the cause of the problem and eliminate it.

4 Implementation

Most of the virtual categories that are displayed in our browser can be computed straightforwardly. The exception is the *requires* set. In this section, we therefore focus the problem of computing the *requires* set efficiently enough for it to be displayed in real-time.

To compute the required methods of a class we must consider all of the class's *reachable* methods, that is,

(1) all methods that are locally defined in the class,
(2) all non-overridden methods defined in its superclasses, and
(3) all methods that may be reached by super-sends from other reachable methods.

The requires set of a class contains all of the message selectors sent to self in one of

the reachable methods, minus the selectors of the methods really provided by the class (and by its superclasses). A method is really provided by a class if there is a real implementation; marker methods such as self subclassResponsibility, and self shouldNotImplement do not count as real implementations. These definitions have been formalized elsewhere[13].

In order to compute this set, we first must find the self-sends and super-sends of a method. Whereas the super-sends can be immediately retrieved from the byte-code, computing the self-sends is more complicated, because they do not all emanate from a single syntactic construct. Consider, for example, the following method.

```
fasten
    | anObject |
    self hook.
    anObject := self.
    anObject button.
    self class new clip.
```

From the code of this method, it is immediately clear that hook is sent to self. What about button and clip? These messages are also sent to an instance of the current class, and so they are really self-sent too. However, detecting this requires a deep analysis of this method, as well as of the method new on the class side.

Our current implementation does not carry out such an analysis. This means that in the above method, hook is the only self-send that we would detect. We compensate for this deficiency by allowing the programmer to declare *explicit requirements* by implementing a marker method with the body self explicitRequirement.

Even with this simplification, computing the requires set in real-time is quite challenging. The main problem is that a single change in a class may affect the requires set of all its subclasses. Thus, a change in Object may mean that we have to update the required methods of all of the classes in the system. A naive implementation based on the above definition would be far too slow to provide the programmer with useful feedback (see section 4.4 for a performance comparison). Updating the requires set in real-time required an optimized algorithm that caches critical data and takes advantage of the coherence of the inheritance hierarchy.

4.1 Caching Self-Sends and Super-Sends

In an early version of our implementation, we used a special parser to extract the self-sends and super-sends from the source code of a method. Because this parsing process is quite time-consuming, we inferred these sends only when a method was first created and cached them.

When computing the requires set, looking for methods that contain a certain self-

send is far more common than looking for (or modifying) the self-sends of a particular method. Therefore we index the caches by the sent selectors: for each class C we maintain a dictionary whose keys are the selectors that are self-sent by the methods directly implemented in C, and whose values are the array of selectors that name the methods that perform those self-sends. For example, if the selector x is sent to self by the local methods a and b, looking up x returns the array #(a b).

Super-sends are critical for determining the set of reachable methods, so we also maintain a cache of the super-sends that are issued by the local methods of each class; this cache is similarly indexed by the super-sent selectors rather than by the selectors of the methods that perform the super-sends.

In our current version, we extract the self-sends and super-sends by abstract interpretation of the byte-code, which is more than 50 times faster than the approach based on parsing the source code. Nevertheless, we still maintain the caches because they make the information available in a form that is better suited for the needs of our algorithm. In section 4.5, we show how the algorithm can be modified so that it uses smaller caches and takes advantage of this faster way of computing the self-sends and super-sends.

4.2 Using the Coherence of the Inheritance Hierarchy

When a method is added, modified or removed, we need to check its class, and all its subclasses, to see whether there is any effect on the requirements. The heart of this computation is checking whether a given selector is self-sent in a given class. Especially in the case of large hierarchies, performing this check separately for each subclass proved to be far too slow. Therefore, we developed a recursive algorithm that takes advantage of the coherence that typically exists between neighbouring classes in the inheritance hierarchy.

The main method of our algorithm is applied initially to the topmost class C; it ascertains recursively, for C and each of C's subclasses, whether that class self-sends a given selector x. As the method proceeds down the inheritance hierarchy, it remembers a method that self-sends x (if one exists). Unless this method is overridden in the next subclass C', we immediately know that C' also self-sends x. Otherwise, we call a helper method that tries to find another method that contains a self-send to x and is reachable from this subclass. The helper method first searches the local methods of C', and then recurses up through all the superclasses until it either finds such a method or reaches the top of the hierarchy. While going up the inheritance hierarchy, we maintain the set of all the unreachable methods in order to avoid false positives.

We now give a more detailed description of both the main method and the helper method.

Main method. The main method of the algorithm searches through the argument class C and all its subclasses and ascertains which of these classes self-send the selector x. In addition to C and x, this method also takes an argument y, the selector of a method that is known to issue a self-send to x in the superclass. This argument is **nil** if there is no such method in the superclass or when the main method is called on the first class of the hierarchy. The main method proceeds as follows.

(1) If y is not **nil**, we check whether y is overridden in the class C.
(2) If y is overridden or **nil**, we call the helper method discussed below to search for the selector of a reachable method that self-sends x. The result is stored as y′ and is **nil** if no such method exists.
(3) If y′ is still **nil**, we mark C to indicate that it does not self-send x. Otherwise we indicate that the class C does self-send the selector x.
(4) We perform a recursive call for each of the direct subclasses of C, passing y′ as a parameter.

Helper method. The purpose of the helper method is to find a methods that self-sends the selector x and is reachable from the class C. In addition to C and x, this method takes a third argument U, the set of all selectors that have been found to be unreachable. This set is empty when the method is called from the main method. The computation proceeds as follows.

(1) We check whether the class C contains a local method that issues a self-send to x and whose selector is not in the set U of unreachable selectors. If we find one, we return its selector and exit.
(2) Before we use recursion to search the superclass, we check whether the superclass has been marked as having no self-send to x by the main method. If so, we return **nil** and exit.
(3) We construct the set U' of all the superclass selectors that are unreachable from the class for which this helper method was initially called. Since all the unreachable selectors in C remain unreachable, the set U' includes all the selectors in U. In addition, U' contains all the selectors for which methods are defined in C (and which therefore have the potential to override superclass methods) and which are not super-sent in C.
(4) We use recursion to find a superclass method that contains a self-send to x. In order to avoid finding unreachable methods, we pass the set U' as a parameter. The result is stored as y and is **nil** if no such method exists.
(5) If y is not **nil**, we check whether y is reachable by a super-send. If so, we update y to so that it names the local method that performs the super-send. Then, we return y.

4.3 Discussion

As a validation, we have used our algorithm to compute the self-sending classes for each of the 12 542 selectors in our Squeak 3.2 image and compared the results to the ones obtained with a naive implementation. During this process we also gathered a lot of profiling data that we used to optimize our algorithm. We now briefly discuss some of the insights we gained during that optimization process.

Choosing a lazy approach In both the main and the helper method, we use a lazy approach: we stop as soon as we identify a single method that self-sends the given selector. In the main method, for example, we keep only a single method that is known to issue a self-send in the superclass, even though there may be other methods that could be cheaply retrieved from the caches. The reason for this is that overrides of the single known method are extremely rare: in our image they occur less than once in 100 000 calls of the main method. It is therefore not even worth the relatively small overhead of maintaining a set of selectors.

Design of the caches The caches are well-suited to the algorithm. In step (1) of the helper method, we can find all the local methods that issue a self-send to x by a single lookup in the self-send dictionary. Similarly, in steps (3) and (5) of the helper method, we are able to check whether a selector is reachable via a super-send by a single lookup in the super-send dictionary.

The caches can also be kept up-to-date quite cheaply. This is mainly because the caches contain only local data, that is, the the cache for a particular class does not depend on other classes in the hierarchy. Thus, modifying a class requires updating at most the local cache for that class. Furthermore, changing the place of a class in the hierarchy does not affect the caches at all.

4.4 Performance Comparison

Our first implementation of the browser deployed caches for the self-sends and super-sends of every method. However, unlike the approach presented above, these caches were indexed by the selectors of the methods performing the self-sends rather than by the selectors that were sent. Furthermore, our initial algorithm did not take advantage of the coherence in the class hierarchy. This meant that the method for finding out whether a class C self-sends a selector x was applied to each class separately. Using this implementation to find out which classes in the system required the selector + took several minutes (188 seconds), [3] which made it impossible to provide real-time feedback.

[3] All the performance data provided in this paper were measured in a Squeak 3.2 image consisting of 1860 classes (not counting the meta-classes), and were executed on a Mobile Pentium III 1.2GHz with 512MB RAM.

In a second version of the algorithm, we used the same caching strategy, but took advantage of the coherence in the class hierarchy. Performance improved significantly, but the same computation still took over 9 seconds. Finally, using the caching strategy and the algorithms presented above, the same test takes less than 50 milliseconds and thus meets our requirement for instantaneous feedback.

4.5 Optimizing the Algorithm for Smaller Caches

The self-send and super-send caches of all the 3638 classes in our Squeak 3.2 image contain about 25 000 sent selectors and 50 000 senders. (Because each class has its own cache, there are far more cache entries than there are distinct selectors.) This means that the caches occupy about 750 kilobytes of memory, or about 8% of the image. To reduce this overhead, we developed and implemented a modified version of this algorithm that uses a space-optimized caching strategy.

The basic idea is that we do not cache the super-sends and cache only a single sender for every selector that is self-sent in a class. This means that our caches consist of about 22 000 entries that associate sent selectors to senders and therefore occupy about 370 kilobytes, less than half of the original size.

Even without changing the algorithm, this causes a performance penalty of less than 15% for 99% of the selectors. This is because the single senders either are (a) not overridden in the subclasses, or are (b) overridden by subclasses that contain another local sender anyway, or are (c) located in classes that have so few methods that searching all of them for another sender is very fast. However, for the remaining 1% of selectors, the computation can take up to 2 500 milliseconds in the worst case, which is about 40 times longer than before.

In order to reduce the computation time in these cases, we introduced some temporary caches into our algorithm. These caches store information about which methods of a class have already been searched for a self-send and which of these methods actually contain a self-send. In this way we can avoid the situation where multiple subclasses override the single sender that is cached for the superclass and therefore cause all the methods of the superclass to be searched multiple times. Using these temporary caches the computation takes a maximum of 400 milliseconds and is therefore only about 6 times slower than the original algorithm with full caches.

This strategy can be further optimized by caching the selectors of all the sender methods in those classes that have many selectors. Alternatively, we can be clever when choosing which one of the senders should be cached: we just choose the method which is least overridden in the subclasses.

The latter approach is very effective, but it depends on inter-class information, and

it is therefore hard to ensure that the caches remain in this optimized state. An effective way of achieving this is to update the caches while executing our algorithm. This means that we use the information from our temporary caches to update the permanent caches, thus making future executions of the algorithm for the same selector much faster.

Using these optimizations allows us to further reduce the computation time so that even in the worst case it takes less than twice as much time as the original algorithm, while the size of the permanent caches is halved.

5 Related and Future Work

The idea of keeping an automatically updated list of things that remain to be done dates back at least as far as the "grass catcher" of Trellis [14], and has been adopted in some form or other in many IDEs; the "Tasks" window of Eclipse is another example. However, such lists are typically created as a by-product of a global re-compilation, rather than being constructed modularly as the consequence of a change to a single method, as in Trellis and in our virtual categories.

Recently, there have been other extensions to Smalltalk browsers that provide the programmer with automatically updated information about the code. In the introduction, we mentioned that there are Smalltalk implementations such as VisualWorks that decorate method names in the browser to indicate things such as supersends or overrides. However, these browsers neither use this information to group methods into virtual categories nor compute the set of required methods.

SOUL, the Smalltalk Open Unification Language [15], is an open, reflective logic programming language written in VisualWorks Smalltalk. Although it is not itself a browser, it can be used to extend a Smalltalk browser with many kinds of logical reasoning, triggered by the occurrence of an action such as accepting a method. This ability has been used to formulate queries that look for composite patterns, and to infer the types of instance variables. SOUL has also been used to create "virtual classifications" that map source-level artifacts to higher-level architectural components (and vice versa) [16].

The Refactoring Browser adds many useful features for semi-automated refactoring to the standard Smalltalk browser, and is available for several Smalltalk implementations including Dolphin Smalltalk [17], VisualWorks, and Squeak. Refactorings supported by this browser include moving methods up and down the inheritance hierarchy, extracting a block of code into a method, and renaming instance variables and methods. The Refactoring Browser uses type inference to help the user to decide which senders actually refer to the method being renamed

Whisker is a Smalltalk browser implemented in Squeak. Whisker's main contribu-

tion is a screen layout that provides a simple and intuitive way to view the contents of multiple classes and methods simultaneously, while using the screen efficiently and avoiding the need to manually move and resize windows. Whisker does this by using subpane stacking, *i.e.*, dynamically stacking subpanes into a single column. Whisker also infers and displays information about the classes of objects that are bound to the instance variables of a class.

Another extended browser is the Star Browser [18], which is available for Squeak and VisualWorks. Like Whisker, the goal of the Star Browser is to allow the programmer to establish a working context without having to deal with too many windows. Unlike Whisker, the Star Browser is built on top of a lightweight classification model that allows one to categorize any sort of item such as classes, methods, method categories, *etc.*. Besides *extensional classifications* that are just bags of items, the model also allows one to express *intentional classifications*, which are defined by a description, and whose contents are updated automatically.

Despite these efforts, we believe that there remain many opportunities to improve the Smalltalk programming environment. One example is applying the techniques of this paper to provide the user with always up-to-date information about the collaborations between classes in aggregation and delegation relationships. We believe that identifying the key information necessary to understand these collaborations could lead to tool and process improvements similar to those that we have observed in the case of inheritance. For example, a browser might support a virtual category showing the methods that a class requires in order that its instances understand all the messages delegated to them.

However, the process of actually computing this information will be much harder because, unlike inheritance, aggregation and delegation are dynamic collaborations. Especially because Smalltalk is not statically typed, it is not immediately clear which instance variables and method arguments may refer to instances of which classes. Nevertheless, this problem can be tackled by combining the results of type inference with information about the classes of instance variables and method arguments that is gathered dynamically. Although this will not always lead to completely accurate information, it seems likely that approximate information will be quite adequate in practice.

6 Conclusion

We have identified four virtual categories that, if updated in real-time, provide valuable insight on the program under development. This categorization structures the space of methods in a way that is quite different from the explicitly declared protocols, and in doing so reveals a different view on the code. Whereas the protocols group the methods according to their role in the domain logic (*i.e.*, testing, printing, model access, *etc.*) the virtual categories group the methods according to their role

in the composition, that is, in the way that the class interacts with its neighbours in the inheritance hierarchy.

Our experience with the extended browser has shown that this view is very valuable both for writing and for understanding the code. While writing, it supports an incremental style of programming: the programmer can freely compose components and add methods, and rely on the browser to maintain an overview of what still remains to be done and where possible problems (*e.g.*, open requirements and overrides) might lie. Later, the same view helps the programmer to understand the code, because at a glance she can see all the critical methods that are essential for understanding the interaction between the various components. This stands in contrast to the conventional viewpoint, which leaves the programmer the task of finding these critical methods by looking through many methods (sometimes hundreds), spread over many protocols.

From an implementation point of view, most of these categories are quite easy to compute. The exception is the computation of the requires set. There are three reasons for this. First, the absence of explicit type information makes it hard to detect the requirements; second, finding whether a method is reachable turns out to be non-trivial; and third, heavy optimization is required to obtain real-time performance.

Nevertheless, our experience has shown that these obstacles can be overcome. Although we based our analysis on a very simple definition of self-send, in practice most of the requirements are detected. Furthermore, caching information about the requirements means that the requires set can be re-computed quickly, even after a change that affects many classes.

Acknowledgements This work has been partially supported by the Swiss National Foundation, by the J.M. Murdock Charitable Trust, and by the National Science Foundation of the United States under awards CDA-9703218, CCR-0098323, and CCR-031340. The authors are also indebted to their colleagues at OGI and SCG for their insights and feedback on this work.

References

[1] 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.

[2] Niraj Jetly. VisualAge for Java 2.0. *Java Developer's Journal*, 4(4):48–49, April 1999.

[3] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

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

[5] 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*, pages 318–326. ACM Press, November 1997.

[6] Squeak home page. http://www.squeak.org/.

[7] Cincom Smalltalk, September 2003. http://www.cincom.com/scripts/smalltalk.dll/.

[8] Nathanael Schärli. Traits — composable units of behavior, September 2003. http://www.iam.unibe.ch/∼scg/Research/Traits.

[9] Daniel H. Ingalls. Design principles behind Smalltalk. *Byte*, 6(8):286–298, August 1981.

[10] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[11] Larry Tesler. The Smalltalk environment. *Byte*, 6(8):90–147, August 1981.

[12] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, November 1986.

[13] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[14] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian. The Trellis programming environment. In *Proceedings Object-Oriented Programming Systems, Languages and Applications, (OOPSLA'87), ACM SIGPLAN Notices*, volume 22, pages 91–102. ACM Press, October 1987.

[15] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[16] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, June 1999.

[17] Doplhin Smalltalk, September 2003. http://www.object-arts.com/DolphinSmalltalk.htm.

[18] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages, Systems and Structures*, 2003. (To appear, special issue on Smalltalk).

Nathanael Schärli is a PhD student in computer science at the University of Bern, where he is a member of the Software Composition Group, led by Prof. Oscar Nierstrasz. His research is in the field of programming language design and implementation.

Before starting his Ph.D. in 2001, Schärli received his master's degree in computer science, also from the University of Bern. While studying for his Masters, he served as an intern in Alan Kay's research group, where he developed a handwriting recognizer for Squeak Smalltalk. Since then, Schärli has continued to collaborate with Kay's group on handwriting recognition and programming language research.

Andrew Black is a Professor of Computer Science at the OGI School of Science & Engineering, which is part of the Oregon Health & Science University. Prior to joining OGI as Head of Department in 1994, Black was a researcher with Digital Equipment Corporation, and prior to that an Assistant Professor at the University of Washington. He holds a D.Phil. from the University of Oxford, where he studied under Professor C.A.R. Hoare.

Black's research interests are in programming language design, distributed systems, and software engineering. He was a co-designer of the distributed object-oriented programming language Emerald, and has published extensively on objects, distribution, and the relationship between them.