Tackling Software Navigation Issues of the Smalltalk IDE

David Röthlisberger
Software Composition Group,
University of Bern
roethlis@iam.unibe.ch

Oscar Nierstrasz
Software Composition Group,
University of Bern
oscar@iam.unibe.ch

Alexandre Bergel
Computer Science Department
(DCC), University of Chile
abergel@dcc.uchile.cl

Stéphane Ducasse
INRIA-Lille Nord Europe
stephane.ducasse@inria.fr

Abstract

The IDE used in most Smalltalk dialects, including Pharo, Squeak and Cincom Smalltalk, did not evolve significantly over the last years, if not to say decades. For other languages, for instance Java, the available IDEs made tremendous progress as Eclipse and NetBeans illustrate. While the Smalltalk IDE served as an exemplar for many years, other IDEs caught up or even overtook the erstwhile leader in terms of feature-richness, usability and code navigation facilities. In this paper we first analyze the difficulty of software navigation in the Smalltalk IDE and second illustrate with concrete examples the features we added to the Smalltalk IDE to fill the gap to modern IDEs and to provide novel, improved means to navigate source space. We show that thanks to the agility and dynamics of Smalltalk, we are able to extend and enhance with reasonable effort the Smalltalk IDE to better support software navigation, program comprehension, and software maintenance in general. One such support is the integration of dynamic information into the static source views we are familiar with. Other means include easing the access to static information (for instance by better arranging important packages) or helping developers re-locating artifacts of interest (for example with a categorization system such as smart groups).

Keywords development environment, source code navigation, software analysis, visualization

1. Introduction

Object-oriented systems form a large space containing plenty of source artifacts such as classes or methods. Often conceptually related code is scattered and distributed in this source space, thus locating the correct places responsible for certain software features is challenging. Object-oriented language features such as polymorphism, inheritance and method dispatching make it very hard to discover the appropriate code in terms of classes and methods purely based on the static software structure. To understand such software features developers also need to have access to dynamic information. Having available behavioral information supports developers in maintaining, extending and evolving software products [Röthlisberger et al. 2008].

The primary tool used by developers to reason about software systems is the integrated development environment (IDE). It supports the navigation of static source artifacts with dedicated browser facilities such as source code trees to navigate from packages to classes and methods. In addition, IDEs usually encompass tools to access software dynamics: debuggers to follow the execution flow of a system, inspectors to study objet state and profilers to analyze the efficiency of specific system executions.

However, the ever-increasing complexity of software systems requires us to think about novel means to navigate large software spaces. In recent years, the Smalltalk IDE [Goldberg 1984] as contributed by dialects such as Squeak [Ingalls et al. 1997], Pharo[1] and Cincom Smalltalk [VisualWorks] did not make significant progress in this regard. IDEs for other languages like Java (*e.g.,* Eclipse [Eclipse03a] or NetBeans[2]) on the other hand significantly improved their source space navigation facilities over the last few years. For example, Eclipse provides Mylin [Kersten and Murphy 2005], a tool analyzing the navigation history to visu-

[1] `http://www.pharo-project.org/`

[2] `http://www.netbeans.org/`

ally give feedback to developers about the importance of software artifacts for the task-at-hand. These visual clues help developers to quickly navigate software systems in a more accurate fashion when solving maintenance tasks. NavTracks [Singer et al. 2005], another tool available for Eclipse, supports software maintenance by recommending, while looking at a specific class file, related files containing classes relevant for the task-at-hand. Thanks to NavTracks' recommendation list, developers can quickly identify source artifacts that are likely to change in tandem with the currently selected artifact.

The history of Smalltalk clearly highlights its power to serve as a platform, environment and testbed to experiment with novel IDE facilities and techniques for software navigation [Goldberg 1984, Ingalls et al. 1997]. For this reason, we implemented several ideas for a better and enhanced IDE in Smalltalk. Concretely, we extended the OmniBrowser IDE of Squeak and Pharo [Bergel et al. 2007] with a number of novel and innovative facilities aimed at easing software navigation and software maintenance.

In this paper we first discuss the problem of software navigation in the Smalltalk IDE. Second, we explain and present in Sections 3 to 7 our extensions to the OmniBrowser IDE to mitigate these navigation issues. In these sections, we also report on the validation we performed for some of these techniques. Section 8 wraps up the paper with some concluding remarks and future directions for further work.

2. Navigation Problem

Software navigation is a crucial prerequisite for program comprehension. To gain an understanding for an unfamiliar software system, developers usually use their development environment. We analyzed several recorded development sessions to reveal how well this process is currently supported in the traditional Squeak IDE. We further analyzed these sessions to elicit ideas for the improvement of software navigation. In this section we report on the findings of this study in terms of navigation issues and opportunities to overcome these issues.

Most software systems spread their functionality over multiple source artifacts. Even reasonably sized systems contain several hundreds of these artifacts (classes, methods). As conceptually related code is often distributed over the entire source space, understanding for instance a particular software feature requires developers to spend considerable time and effort to navigate this feature. During the navigation, it often even happens that developers lose the context or the overview and have to start over searching for the right path to be able to comprehend a software feature. In the following, we illustrate these navigation problems with some indicators obtained from empirical studies.

Problem indicators. As indicators for navigation difficulties we consider the number of *window switches* (changing focus from one window to another), the number of *re-visits of source artifacts* purely for reading and understanding (without modification), *edit/navigation ratio* (ratio of edit actions compared to navigation actions), *the extent of navigation until first edit* (how many navigation actions a developer performed until modifying the first artifact) and *the average extent of navigation between two edits* (how many navigation actions occur between two subsequent modification actions). By analyzing 20 development sessions we obtained the results displayed in Table 1 for these five indicators. All these recorded and analyzed sessions originate from developers working for 30 minutes on software maintenance tasks in small or medium-sized applications with up to hundred classes in Pharo and Squeak Smalltalk.

Indicator	Average of 20 sessions
Number of window switches	38.85
Number of entities revisited	35.10
Edit / navigation ratio	9.51%
Number of navigation actions until first edit	52.14
Number of navigation actions between two edits	19.31

Table 1. Five indicators highlighting navigation issues occurring in the Squeak Smalltalk IDE

The concrete numbers for the various indicators obtained in this survey confirm the hypothesis that navigating the source space in Smalltalk is often difficult. Developers frequently have to switch between different windows; opening views on the same source artifacts several times is also a frequent incident, even in short development sessions lasting for just half an hour. Moreover, developers usually spend quite some time until they are able to locate in a maintenance task the artifacts they actually want to modify to correct a defect . The edit/navigation ratio is very low, with less than ten percent. All these figures demonstrate that for the particular sessions we studied, the extent of navigation activity required to identify an artifact to be changed is large. This is in particular true for the beginning of a task when developers perform on average 52 navigation actions before they locate the first entity they want to modify. Another indication for a possibly ineffective navigation in IDEs is the high average number of navigation actions performed between two subsequent modification actions (on average 19 navigation actions).

Problem identification. Firstly, from the numbers shown in Table 1 we conclude that software space navigation is an important development activity that takes a considerable amount of time. Secondly, we want to extract from these numbers, as well as from interviews and discussions we had with developers, the concrete reasons why the IDE does not better support the navigation of software systems. We identified five main reasons why the IDE at its current state ineffectively supports navigation:

- *Working set, context representation.* The working set, that is, the entities the developer is currently working with, is not appropriately represented and maintained in the IDE.

The IDE just opens views on source entities in various windows, but does not maintain any connection or references between these windows, thus it is not clear which window belongs to which working set or task. The developer hence cannot identify the context in which he opened a particular window. There is no mapping between windows and activity or task being performed. Up to now, the IDE did not even provide a navigation history to see what has recently been navigated in a window to help the developer determining the activity in which this window was opened originally.

- *Distributed and distant artifacts.* Conceptually related source artifacts are often distributed over the software space and distant to each other (for instance in different packages), but the IDE does not support the easy navigation of distant entities. Packages are sorted alphabetically in the package tree; software systems spread over various packages could thus be distributed over the entire package tree, requiring the developer to scroll up and down the tree while navigating the source code.
- *Implicit and hidden dependencies.* Many dependencies and relations between different source artifacts are not directly visualized in the IDE. To for instance reveal the classes used by a package or another class, developers would have to scan every single method to find out all collaborating classes. Similarly, packages used by or dependent on a particular package are also difficult to reveal as the IDE does not provide direct support to present such dependencies. Even when dependencies are explicit (such as super- / subclass relationships), the navigation from one to the other is often still not easily possible. For instance when having selected an overriding method in a subclass, navigating to the superclass to study the overridden method requires the developer to manually locate and select the method in the superclass again.
- *Behavioral information unavailable.* Many dependencies between source artifacts are not visible in the static source code at all. Without using debuggers or inspectors, developers cannot see these dependencies. But using these tools requires developers to manually set breakpoints and to leave the familiar source code browser in favor of a separate tool. However, the IDE itself could enhance the representation of the source code with runtime information about types and dependencies. The presentation of runtime information in its static views enables the IDE to make developers aware of runtime dependencies and thus to provide means to better navigate them, for instance to follow the execution flow from one method to the other.
- *Window plague.* As IDEs show source entities in windows or tabs, normally each artifact is opened in new window. This quickly leads to a cluttering of the workspace with many windows, in particular in Pharo and Squeak. If the developer does not take the time to manually close windows, the number of open windows steadily grows, thus further worsening the navigation problem as developers have to spend considerable time to locate windows of interest, some of them might even be hidden behind others. However, manually closing windows is often difficult as developers cannot be sure which windows they will still need in the future.

Problem solution. In the subsequent sections, we introduce possible solutions mitigating the aforementioned problems. For each problem, we discuss two or three techniques we have implemented to help developers better navigating the software space in the IDE and to address a particular navigation issue. For some techniques we also report on some empirical feedback or other evaluation we conducted to assess the usefulness and extent of improvement of this technique resulting in practice.

3. Representing Working Sets

To better represent working sets and context in the Smalltalk IDE, we implemented several techniques such as Smart Groups which developers can use to categorize artifacts in working groups, for instance in groups for particular defects they need to fix. A means to open multiple methods enables developers to group together in a browser all methods they need to understand and/or adapt for the completion of a certain task. Having available the full history of all viewed source artifacts for each browser enables developers to go back and forth in their navigation history to reconsider their work in their particular development context. The integration of test execution facilities makes it possible to embed the test running process in the current working context, that is, without having to open another tool or window such as a dedicated test runner.

3.1 Smart Groups

Smart groups are a categorizing mechanism orthogonal to the standard categorization applied in Squeak which is based on (static) packages. Smart groups are displayed in the same column as packages and the hierarchy view. Developers can themselves create categories as they wish and place any source artifact (package, class, method, class category, etc.) in one or several smart groups. One smart group could for instance hold all classes and methods implementing a logging feature of an application, and another smart group could contain the artifacts defining a HTML rendering feature. Thus smart groups make distributed source artifacts accessible under a manually give name such as “logging”.

Besides the manually managed smart groups, we support also automatically created groups that hold results of submitted search queries. At the top of the browser there is the so-called mercury panel accepting search queries for classes, methods, class references, senders, implementors, etc. After submitting such a query, the developer obtains the re-

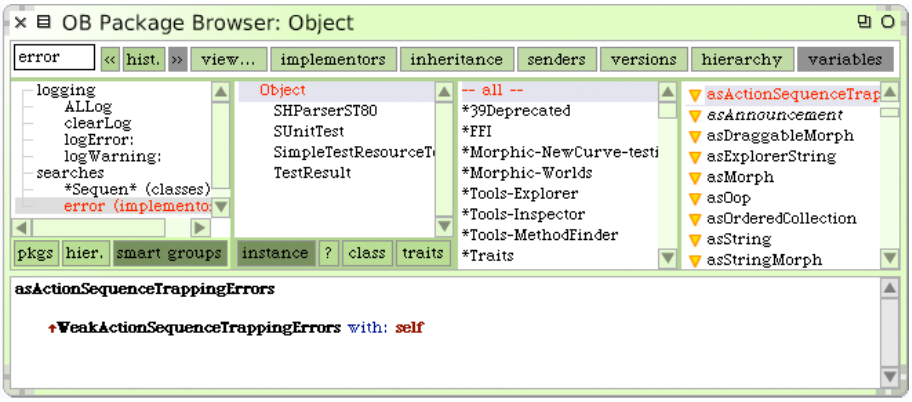

Figure 1. Smart Groups: Manually created logging group and automatically created search groups.

sult in a smart group named after the search query. Whenever this group is selected, the query is processed again, thus the search results are always accurate. Both types of smart groups are illustrated in Figure 1.

Smart Groups are a similar concept as available in early versions of Smalltalk when people were not working with packages to maintain code, but with change sets. A particular change set can be considered as a smart group bundling code from different classes and categories, that is, code not statically related by means of hierarchy or containment. A similar concept is also applied by Intensional Views [Mens et al. 2006]. An intensional view is an executable description which yields a set of entities belonging to the views; we refer to these entities as the extension of the view. The Star Browser [Wuyts] allows the developer to classify during navigation any source entity in different categories; the classified source entities can be accessed under these classifications.

3.2 Opening Multiple Methods

We support the display of several methods at a time in the code panel. Developers can open several methods and see their code in a row in the code panel, to for instance compare their implementation. By pressing the command key, a developer decides to open a method in an additional view instead of browsing the method in an already existing method view. Each method view can be pinned, moved, or closed. Pinning means that the method in this view is fixed, that is, it will not be overwritten with another opened method. If there are only pinned method views open, browsing a new source entity always triggers the opening of a new view. Changing the order of the method views is possible by dragging and dropping the dedicated move button on the move button of another view. New methods are always viewed in the right most method view, if this one is not pinned. Otherwise, a new method views will be added to the code panel on the right.

Viewing multiple methods at the same time and being able to edit them independently is particularly useful when working with conceptually related code scattered over different locations and classes. Developers are able to open all methods that for instance implement a particular feature and can study and alter their implementation at the same time in the same browser window, even though these methods come from different classes or even different packages.

Other environments or dialects related to Smalltalk already provide the possibility to edit multiple source entities at the same time, for instance Strongtalk [Strongtalk], Newspeak[3] [Bracha 2007] or Whisker [Way 2005].

3.3 Navigation History

As an extension to the browser, we provide history functions such as back/forward button and full history access as a drop down menu, similar to the way web browsers show the history of visited web pages. Each source artifact we navigated in a particular code browser is accessible in the browser's history, each entry in the full history also comes with information about its type (package, class, method, protocol, etc.) to ease the reading of the history list, see Figure 2. Back and forward buttons always navigate to the artifact we browsed before or after the currently active point in history.

The developers we interviewed about the effect of these history facilities reported us that they frequently take use of this feature and benefit from its availability in two main scenarios: First, when they explored a branch of the source space that turned out to be a dead end, they can go back in the history to the artifact from where they started the exploration. Second, the history also gives an overview of what they already explored in the past, which helps them to quickly reconstruct the browser's working context and in particular also to identify important artifacts. Hence an accessible navigation history also supports program comprehension as it can serve as an overview of, for instance, all artifacts responsible for a certain software feature as they have all been navigated in the past and are thus part of the

[3] http://bracha.org/Site/Newspeak.html

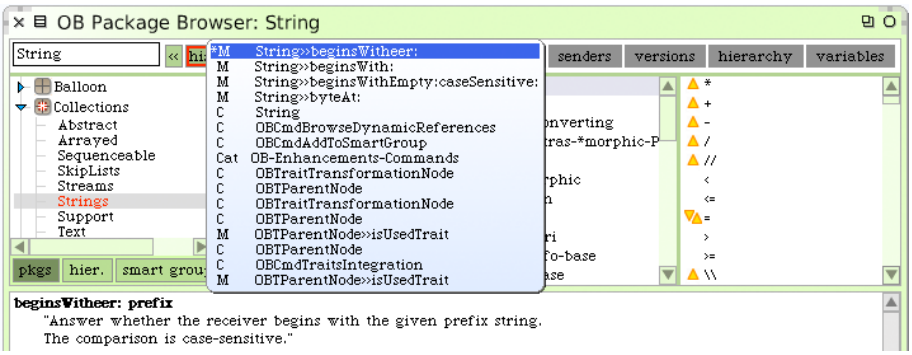

Figure 2. The full history of all navigated source entities in this browser.

navigation history. In general, developers very much appreciate having available the navigation history in their daily development work and in each code browser.

3.4 Test Integration

Running (and writing) tests is an important development activity. To account for this importance we wanted to make the test execution as easy and smooth as possible. For this reason, we extended the Pharo browser to be able to run test methods and entire test cases or even test packages from within the browser. By pressing command and 't', all tests stored in the selected artifact are executed and the results appear on the screen. If the tests fail, the browser asks the developer whether he wants to open a debugger to locate the cause of the failure or error.

We even extended the SUnit testing framework to also store the results of a test run, that is, which test methods passed, failed or raised errors. We exploit this information in the browser to show by means of icons whether a test method or test class progressed (more tests succeed since the last stored run) or regressed (fewer tests succeed since the last stored run). Such a visual clue on the status and progress of fixing an entire test suite is particularly important when working on a huge system which was for instance upgraded to use a new library. With these visual clues we thus want to express how much progress we made in fixing this system and the test suite to comply to the new library. Another example is when migrating a large system from one Smalltalk dialect to the other. An example is the migration of Moose [Nierstrasz et al. 2005] from Cincom Smalltalk [VisualWorks] to Pharo.

4. Navigating Distributed, Distant Artifacts

The techniques presented in this section aim at easing the navigation of distributed and distant source artifacts, for instance entities organized in different packages. The Package View contributes elaborated facilities to prominently display manually selected packages or packages that have been recently modified, loaded or updated. The Hierarchy View allows developers to more conveniently navigate class hierarchies, for instance by changing on the fly the class around which the hierarchy is shown. The automatic selection of source artifacts is another means to save time while navigating. When browsing a class that has a method with the same name as the previously selected one, the browser automatically selects this method in the new class.

4.1 Package View

Smalltalk's browser did not support Monticello packages, but just showed class categories in its first column. We first added support for packages as the top-most source entity and list the class categories contained in a package below it. We then implemented a tree to more conveniently browse packages. Usually a standard Smalltalk image contains up to a hundred packages and several hundred class categores, thus browsing such a long list is time-consuming. As a tree offers capabilities to collapse and extend branches, the employed package tree helps to reduce the time to navigate to a specific package or class category.

We also added features to more quickly locate important packages: Modified or newly loaded or updated packages are automatically placed at the top of the tree. Even further at the top are packages that the developer manually placed to appear prominently, see Figure 3. These two means to influence the order of the packages is considered as very important by developers working on systems consisting of several different packages that might otherwise be spread over the entire alphabetically sorted tree. Additionally, developers can alter the package order by manually arranging packages in a text file containing all packages in the currently defined order.

4.2 Hierarchy View

The traditional Smalltalk code browser do not encompass a dedicated view for class hierarchies integrated in a standard class browser. There was an external hierarchy browser which developers could open in new windows, thus cluttering the workspace with more and more windows.

We hence extended OmniBrowser to come up with an integrated hierarchy view. This view appears in the first navigation column, that is, where the packages are displayed, as

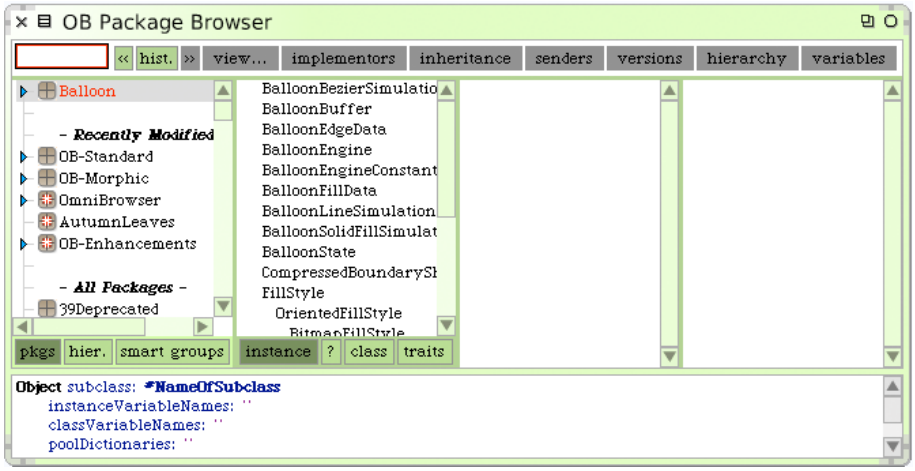

Figure 3. The package view in OmniBrowser with packages place at the top, manually and automatically.

shown in Figure 4. From a selected class we display all its super- and subclasses. Clicking twice on a class in the hierarchy, gives this class the focus, that is, the view shows its super- and subclasses. This feature is useful to easily bring up and navigate an entire class hierarchy. In the hierarchy view, the next column shows all the packages defining or extended the selected class. The defining package is displayed in bold. This feature supports the developer in locating class extensions of particular classes.

4.3 Automatic Selection

To ease navigation of, for instance, class hierarchies, we implemented advanced auto-selection facilities. The browser now remembers the last selected method in a class. If we then navigate for instance to the class' superclass which implements a method with the same name, this method is automatically selected. This auto-selection feature is also active when browsing to statically unrelated classes. If we currently have selected the method size, then the browser will automatically select in each class we navigate the method size, provided that it exists. The browser remembers the selection of size until the developer manually selects another method.

Another automatic selection occurs when browsing a method category with just one method inside; this method is automatically selected. When browsing search results stored in SmartGroups (see Section 3.1), the browser automatically jumps to methods complying to the submitted search query.

Automatic selection improves the efficiency of the developer as it saves him from cumbersomely locating in long method lists of a new class the same method he already had selected before. In particular when navigating hierarchies it often happens that we want to study the implementation of a specific method in different classes.

5. Revealing Implicit and Hidden Dependencies

To make dependencies more explicit in the Smalltalk IDE, we provide a dependency analyzing tool that lists for instance the packages used by a given package by statically analyzing all methods and classes of this package to detect to which other packages it communicates. Icons shown for each source artifact on the other hand can display additional information, for instance, information orthogonal to the static tree structure in which the source code is typically represented. In addition, we use icons to denote methods containing halts, overridden methods, collection classes and dirty packages. The use of traits creates other dependencies such as all the classes using a trait or whether a method in a class is locally defined or stems from a trait. Such dependencies are now explicitly visible in the code browser.

5.1 Dependencies, References

There are various dependencies between different source artifacts, for example a package may have other packages as prerequisite when a class uses other classes to outsource logic and functionality. Many of these dependencies are hidden and not explicit, making them difficult to reveal. For instance, developers often have to study many methods and classes to gain an overview of the various dependencies a package imposes. A package usually not just requires the defined prerequisites, but also communicates to other, standard or non-standard packages, but without explicitly stating them as prerequisites.

Developers of packages are often unaware of these implicit package dependencies. As a consequence, delivering a system based on dependent packages is challenging as system clients may not have the same dependent packages or libraries installed, which ultimately prevents the system from being installed. To be able to check and navigate dependent packages, we provide an analysis tool in the IDE which ana-

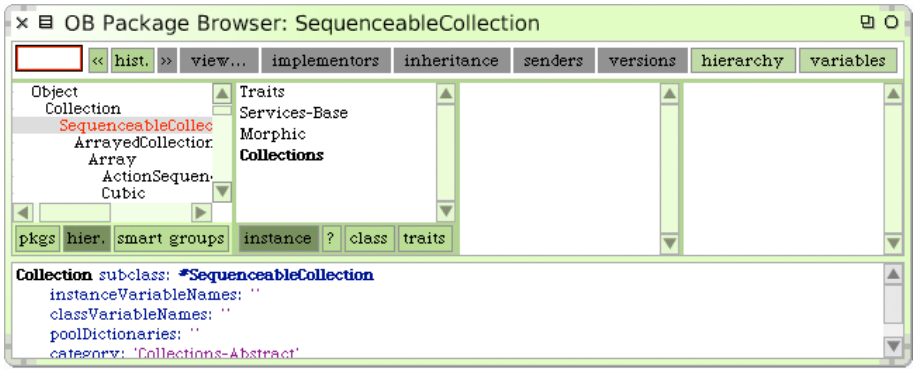

Figure 4. The hierarchy view in OmniBrowser.

lyzes the entire source code of the package to be checked and searches for usages of classes or methods from other packages. All such occurrences are reported to the developer by giving him a list of dependent packages (that is, the packages defining the located external classes or methods used in this particular, analyzed package). For each dependent package the IDE can show all occurrences, that is, the places in source code where a dependency is actually created, for instance in a method sending messages to instances of an external class outside the current package.

In a reference view optionally place next to the source code view we show classes used in the current source artifact viewed by the developer. For a selected method or class, we list instantiated classes, referenced, or classes whose methods are invoked. When Hermion is installed, which is further described in Section 7, we can even display dynamic references, that is, references statically not visible, such as subclasses of a class implementing an invoked method. Statically we are only able to determine a reference to the superclass implementing the method while dynamically we find out that actually a subclass receives the message send at runtime.

5.2 Icons

Icons serve the purpose of visually conveying information that is otherwise not easy to represent, such as information about errors in source code or additional structural information, for instance whether a method is overridden in subclasses. We can use well designed icons to convey richer information than pure text could transfer. Another advantage of icons is that they do not take much space: a 12 pixels square icon conveys valuable information.

For all types of source artifacts we are able to show one icon at a time. If a specific artifact qualifies for more than one icon, we present the one with the highest priority. We mostly use icons to visualize information otherwise not easily accessible. The following list reports on the different icons used for the four main types of source artifacts:

Packages, Class Categories.

- Package icon — to denote whether an entity is indeed a Monticello package
- Published icon — for already published packages
- Dirty icon — packages that have been locally modified but not yet committed
- Newer version — packages with newer version(s) in repository than installed locally

Classes. For classes we visualize with an icon the type of class, for instance an exception class. The developer can himself easily add more class type icons by implementing the method icon in the desired superclass.

- Exception icon — for Exception and subclasses
- Collection icon — for Collection and subclasses

Methods.

- Overridden icon — if a method is overridden in any subclass
- Overriddes icon — if a method overrides the same method from a superclass
- Overriddes icon — if a method overrides and is overridden at the same time
- super send icon — method sending super to the same method
- super send icon, but invoking different super method
- abstract icon — abstract method, that is, one sending isSubclassResponsibility
- halt icon — method containing a halt
- flag icon — method sending flag:
- exception icon — method raising an exception

Test methods, test classes.

- green icon — test method or class running green
- yellow icon — test method or class running yellow
- red icon — test method or class running red

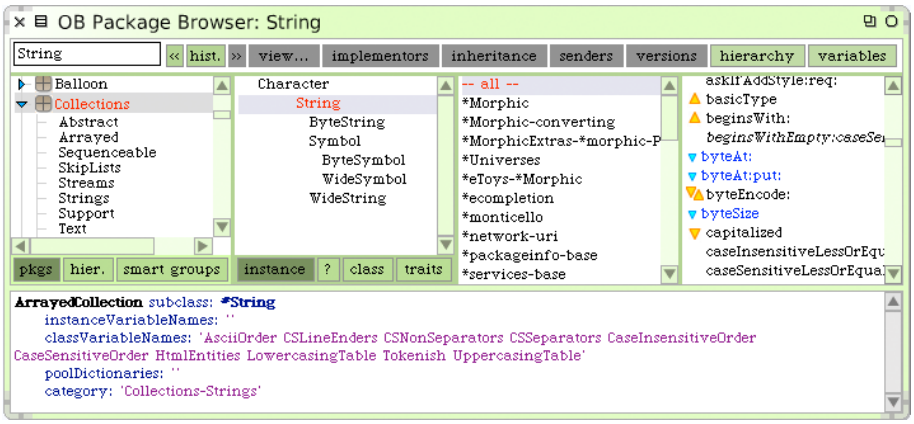

Figure 5. Several method icons appear, such as abstract, overridden, overrides, or overrides and overridden.

- more green than red icon — test class with more green than red running test methods
- equal green and red icon — test class with nearly the same number of green and red running tests
- more red than green icon — test class with more red than green running test methods

Figure 5 shows various method icons appearing for class String. Many icons are even clickable. Clicking on such an icon triggers the execution of an action appropriate for this particular icon. Clicking on test-related icons for instance triggers the running of the tests for which the icon appears. Or clicking on the overridden icon navigates to the method in a subclass overridding the selected method. If several subclasses override this method, then we show a list of classes from which the developer can choose one to navigate there.

5.3 Traits Integration

A trait is a unit of behaviour that can be composed with other traits and used by classes. Traits offer an alternative to multiple inheritance and promote the reuse of methods between unrelated classes [Black and Schärli 2004]. However, the Squeak IDE did not properly support traits and a development process taking use of traits. We improved support for traits in OmniBrowser by implementing various features:

- *Traits used by a class.* Next to the instance, documentation and class button appearing in the class column we put a trait button. This gives all traits used by the selected class in hierarchical order. Developers can study each trait's methods and modify or extend them. This trait view is illustrated in Figure 6.
- *Traits users of a class.* For a selected trait, we provide a command to locate all classes using this trait.
- *Trait methods.* When browsing a class, we see all methods originating from a trait in italic to be able to quickly recognize the fact that we are dealing with code defined in a trait. When re-compiling a method that belong to a trait, the browser asks whether we want to change this method just locally for this class or for the entire trait (affecting other users of this trait as well).
- *Required methods.* In a trait we highlight in blue the methods required to be implemented in classes using this trait.
- *Move to trait.* For each non-trait method we can execute an action called "move to trait" to move this method to an existing trait.
- *Exclude trait method.* When selecting in a class using a trait a method defined in this trait, we can trigger the command "exclude from trait" which with we can either remove the trait method just from this class or from the entire trait.
- *Other trait-related features.* We extended the browser's search facilities to search for traits as we do for classes, using the same tools. Or we implemented sorting algorithms that can hierarchically sort traits as it is done for classes.

This browser extension is very helpful when developing applications using traits. The traditional browser did not represent the relationships between a trait and the classes using it, this means it was not possible to access a list of all users of a trait. Recognizing trait methods in the method list of a class was impossible without double-checking the trait definition. Finding traits by name was also not supported as the search facility covered only classes. Thus it was a huge burden to develop applications using traits. The example of traits highlights the importance of appropriate tool support for the adoption of new techniques. Even when a novel technique clearly brings important benefits, it will not be adopted in practice if tools and development environments do not reasonably support this new technique.

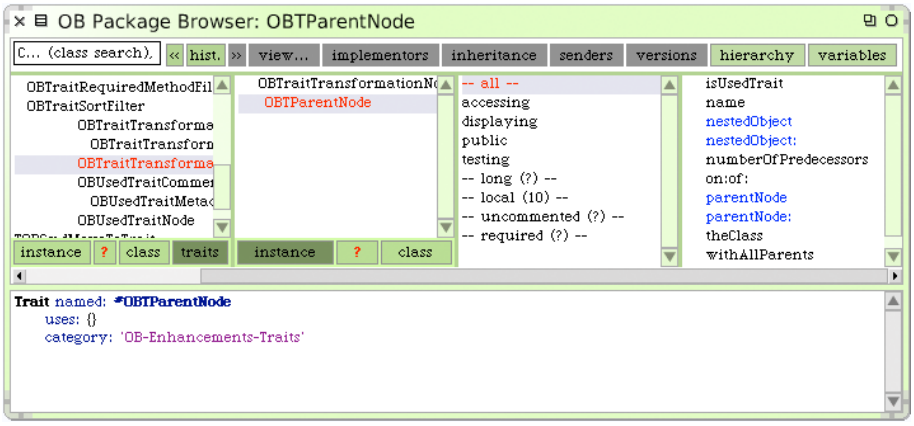

Figure 6. Viewing a class using a trait and browsing this trait in the trait view.

5.4 Source-code Management Integration

Monticello[4] is the standard tool used in Squeak and Pharo to manage source code packages and repositories. Since we provide direct support for Monticello packages, we opted to also add more source code management facilities directly to the source code view. In the package view we added access to various Monticello functions such as committing, updating, viewing history or changes, or importing a new package. Adding support for other source code management systems besides Monticello is straightforward.

6. Integrating Behavioral Information

To further ease software navigation and comprehension we enhance the IDE's static source views with some behavioral information about message sends, variables accesses, or collaborators used in a given class or method.

As stated in the introduction to this paper (see Section 1), it is often impossible to gain a deeper understanding for a software system or parts thereof without also studying its dynamics. While dedicated tools such as debuggers or inspectors can help developers to reconstruct and analyze the execution flow, types of variables and even concrete variable values, or runtime complexity of certain algorithms or objects, the tools primarily used to navigate the source code, the class or package browser or the hierarchy browser, do not encompass information about software dynamics. We consider this a serious hindrance to a more efficient program comprehension and navigation and hence enhanced these IDE tools to also display dynamic information. We refer to the IDE enriched with dynamic information as Hermion [Röthlisberger et al. 2008], which is basically an extended version of the OmniBrowser we referred to in all other sections. Hermion only differs to OmniBrowser by its means integrating dynamic information, otherwise it encompasses all of OmniBrowser's features.

[4] `http://www.wiresong.ca/Monticello`

Concretely, we exploit dynamic information with three basic means in the Squeak IDE:

i) *Message sending.* We enhance the source code view with information about message sending (receiver, argument and return types) and including number of occurrences of each type. For this purpose, we introduce in the source code icons that serve as a navigation aid. Clicking on such an icon appearing right after a declaration of a message send in source code brings the developer to the method that has been invoked at that location. If this is a polymorphic call site, all invoked methods are displayed in a list along with numbers of invocations. The developer can choose the desired callee to navigate there. For message send arguments we provide lists of all occurred types, senders of a method we can navigate by clicking on the back arrow appearing next to the method header. If there are several senders, we see every sender in a list. Figure 7 illustrates how these icons appear in a concrete method.

ii) *Type information.* Similar as for message sends, we also enrich every variable access, read and store, in the source code view with additional information. We display all types that have been bound to a particular variable at this location in code in a list. For each type in this list we give information about how often particular type occurred at runtime. Clicking on a type brings the developer to the definition of the corresponding class. Type information is particularly useful in dynamic languages not having explicit types, thus developers cannot anticipate to what types a variable can be bound at runtime.

iii) *Collaboration information.* As another enhancement, we list all dynamic collaborators of a method or a class, that is, all classes that are referenced in each source entity. This list appears on the right side of the method view. For each referenced class, we are able to obtain all concrete locations in code where this class has actually been referenced, for instance all variables bound to an instance of this class.

To gather the dynamic information on which these three enhancements are based, we use partial behavioral reflec-

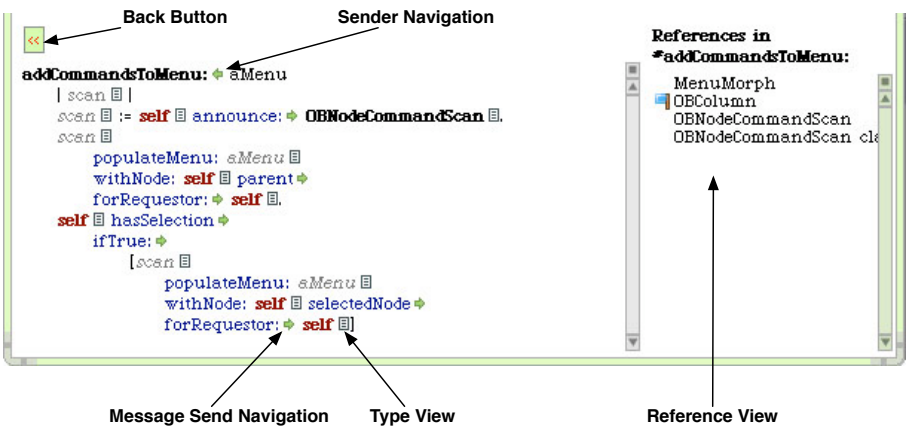

Figure 7. Enriched method source code view including a reference view in Hermion

tion as provided by Reflectivity [Denker et al. 2007]. This approach also allows us to reason about sub-method level operations, for instance to study the types of each temporary variable used in the method body. Another benefit of Reflectivity is the fine-grained selection possibilities it offers; we can freely choose which particular methods or classes, even specific objects, we want to analyze dynamically. These selection capabilities are very important to reduce the overhead coming along with runtime analysis. Currently, we experience a slowdown of factor three to five when we analyze entire systems with Reflectivity to obtain the dynamic information exploited by Hermion.

We gathered feedback from developers about how they personally consider the usefulness of Hermion in their daily work when maintaining software systems. The general feedback was that developers consider it as very useful to be able to reason directly in the source code about software dynamics such as variable types or invoked methods, in particular in a dynamic language that does not declare any types and where it is hence often unclear, what kind of objects are stored in particular variables or which methods are invoked if for instance a selector with a frequent name such as size is used. Further feedback stressed the usefulness of the collaboration view as this helps developers to find both, the classes communicating with a selected class and the locations where this communication is actually occurring.

7. AutumnLeaves - Mitigating the Window Plague

To reduce the number of open windows we worked on a tool and algorithms to automatically identify and ultimately close obsolete windows. This idea and its realization is discussed in this section.

AutumnLeaves are a means to automatically close obsolete, useless windows that will not be needed by the developer in the future, such as code browsers showing methods that are not relevant anymore for the current development focus or inspectors displaying objects no longer being of interest. *AutumnLeaves* analyzes all navigation activities of developers and associates a weight with each open window and each navigated source entity. A window is automatically closed (developers can decline this decision) if its weight drops below a certain percentage of the average weight of all open windows, which is called the closing threshold, usually defined as 30% of the average weight. The weight of a window or an entity is increased on every user action (opening artifacts, modifying code, moving or resizing windows, giving focus to windows, etc.). The total weight of a window is the weight of itself and the weight of the currently displayed entity. Weight is also propagated to statically related artifacts: if we for instance navigate a method, the weight of its declaring entity is also increased. Weights of other related and open entities, such as senders and implementors of this method, will also be increased. Note that *AutumnLeaves* does never decrease the weight of windows or entities; instead weights are always increased and for each window compared to the average weight of all windows. If we would decrease the weight of windows unused for a longer period of time, they would maybe drop out too quickly, for instance if the developer is focusing on one single window during this time. With its weighting mechanism, *AutumnLeaves* basically models the otherwise implicit references between windows and thus identifies the obsolete windows, similar to the way a garbage collector terminates unreferenced objects.

Smalltalk's window management allows windows to overlap each other, we also have to take into account the visibility of each window, that is, whether it is fully visible, partially visible or fully hidden by other windows. A window hidden behind several other windows is much more likely to be not useful anymore, or maybe the developer even forgot about its existence. Thus we reward windows being visible by giving them additional weight points after each user action. The number of weight points given depends on

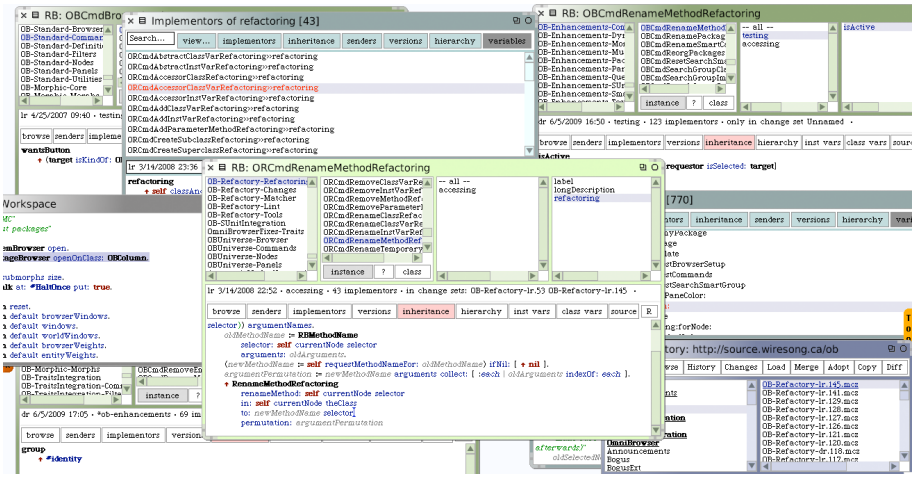

Figure 8. Squeak arranges its windows on a desktop. Windows can overlap each other.

the visible portion of the window. Obtaining the focus also increases the weight of a window, so does moving or resizing or typing in it. Developers can also pin certain windows, those will never be closed by *AutumnLeaves* but always stay open.

AutumnLeaves efficiently reduces the window plague (as depicted in Figure 8) we are typically suffering from in Smalltalk environments. Some experiments we performed to assess the extent of the window plague revealed that in a short development session lasting half an hour, developers performed on average up to 40 switches between different windows and revisited 35 entities again without altering them. These two numbers are a clear indication that developers lose the overview over their workspace as they are confronted with too many windows for which they cannot maintain a clear mental model. From concrete empirical experiments we learned that with *AutumnLeaves* we can remove up to 40% of all open windows, while the same validation showed that developers could have re-used later on only 40% of those windows *AutumnLeaves* suggested to close. As empirical data we analyzed 25 different recorded development sessions concerned with the maintenance or implementation of medium sized object-oriented applications.

8. Conclusions and Future Work

In this paper we first studied the characteristics and the extent of the navigation issues we typically encounter in the Smalltalk IDE when working with large software systems. Taking the identified causes for these navigation issues as a starting point, we elaborated on several different extensions and enhancements we implemented for the Smalltalk IDE to address and at least partially overcome aforementioned issues. These enhancements aim at easing software navigation and at generally improving program comprehension while working with software systems in the Smalltalk IDE. We took the IDE *OmniBrowser* as the basis for our prototype implementation due to its extensible design.

Extension	Problem area
Smart Groups	Working set, context representation
Opening multiple methods	Working set, context representation
Navigation history	Working set, context representation
Test integration	Working set, context representation
Package view	Distributed artifacts
Hierarchy view	Distributed artifacts
Automatic selection	Distributed artifacts
Dependencies, references	Hidden dependencies
Icon	Hidden dependencies
Traits integration	Hidden dependencies
Source-code Management Integration	Hidden dependencies
Hermion	Behavioral information
AutumnLeaves	Window plague

Table 2. Problem area covered and addressed by our IDE extensions.

Table 2 summarizes the results of this paper, that is, the specific problems addressed by the particular extensions presented throughout the paper.

In the future, we will further work on the improvement of the navigation issues mentioned in Section 2. The techniques and enhancements presented in this paper certainly do not completely solve these issues, thus we continue investigating and analyzing how developers navigate software systems in the IDE and want to identify further opportunities how we can improve the navigation experience. After having implemented new enhancements to the source navigation in the Smalltalk IDE, we particularly want to validate the concrete impact on productivity and navigation efficiency resulting from these extensions. This goal also includes the empirical evaluation of the work we already implemented and presented in this paper by means of controlled empirical experiments.

From the implementation point of view, we note that efficiency of the IDE tools are an important issue, in particular in Squeak and Pharo where UI widgets such as trees are not

optimally implemented or where the package management is not cleanly and tightly integrated in the system, yielding serious performance penalties when accessing packages in the IDE. Furthermore, all our extensions are heavily dependent on the underlying framework for user interfaces (Morphic) and browsers (OmniBrowser). This dependency seriously hampers the freedom of choice concerning experimenting with completely new navigation patterns as all these frameworks offer very limited, traditional widgets and concepts how to browse code. Future work thus also aims at entirely rethinking browsing and navigation patterns and concepts by means of developing our one foundation of frameworks to efficiently and effectively do so.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Meta-driven browsers. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 134–156. Springer, August 2007. ISBN 978-3-540-71835-2. doi: 10.1007/978-3-540-71836-9_3. URL http://scg.unibe.ch/archive/papers/Berg07cOmnibrowser.pdf.

Andrew P. Black and Nathanael Schärli. Traits: Tools and methodology. In *Proceedings ICSE 2004*, pages 676–686, May 2004. doi: 10.1109/ICSE.2004.1317489. URL http://scg.unibe.ch/archive/papers/Blac04aTraitsTools.pdf.

Gilad Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.10.004. URL http://bracha.org/executableGrammars.pdf.

Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007. URL http://www.jot.fm/issues/issue_2007_10/paper14http://www.jot.fm/issues/issue_2007_10/paper14.pdf.

Eclipse03a. Eclipse platform: Technical overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984. ISBN 0-201-11372-4.

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 of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997. doi: 10.1145/263700.263754. URL http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html.

Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6. doi: 10.1145/1052898.1052912.

Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Coevolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006. URL http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-26.pdf.

Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. doi: 10.1145/1095430.1081707. URL http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf. Invited paper.

David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3176-2. doi: 10.1109/ICPC.2008.32. URL http://scg.unibe.ch/archive/papers/Roet08bDynamicInfoIDE.pdf.

Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi: 10.1109/ICSM.2005.66.

Strongtalk. The strongtalk type system for smalltalk. URL http://bracha.org/nwst.html. http://bracha.org/nwst.html.

VisualWorks. Cincom Smalltalk, September 2003. http://www.cincom.com/scripts/smalltalk.dll/.

Doug Way. Whisker: The O-O stacking browser, December 2005. www.mindspring.com/~dway/smalltalk/whisker.html.

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