Hermion - Exploiting the Dynamics of Software

Authors: David Röthlisberger, Orla Greevy, and Oscar Nierstrasz
Affiliation: Software Composition Group, University of Bern, Switzerland
Homepage: http://scg.iam.unibe.ch/Research/Hermion
Smalltalk Dialect: Squeak
License: Open-source (MIT)
Keywords: development environment, dynamic analysis, feature analysis

Abstract. The current Squeak Smalltalk IDE provides a structural perspective on a software system in terms of packages, classes and methods. However, from this perspective it is difficult to gain an understanding of how source entities participate at system's run-time. Hermion enriches the traditional IDE with a view on the dynamics of the system, (i) by offering a complementary feature-centric perspective of a software system to allow developers to reason about how specific run-time features of their software are implemented, (ii) by integrating dynamic information into the static perspective on a system, *i.e.,* source code, and (iii) by providing mechanisms to query run-time information.

1 Enhancing the IDE with Dynamic Information

The problems of understanding object-oriented software are poorly addressed by current Squeak development tools, as these tools purely focus on a structural perspective of a software system by displaying static source artifacts such as class categories, classes or methods. The running of the system, however, *i.e.,* the execution of specific system features, is not explicitly represented in Squeak. However, as maintenance requests are typically expressed in terms of features, it is crucial for a software developer to understand the running of features. Dynamic information can enhance this understanding.

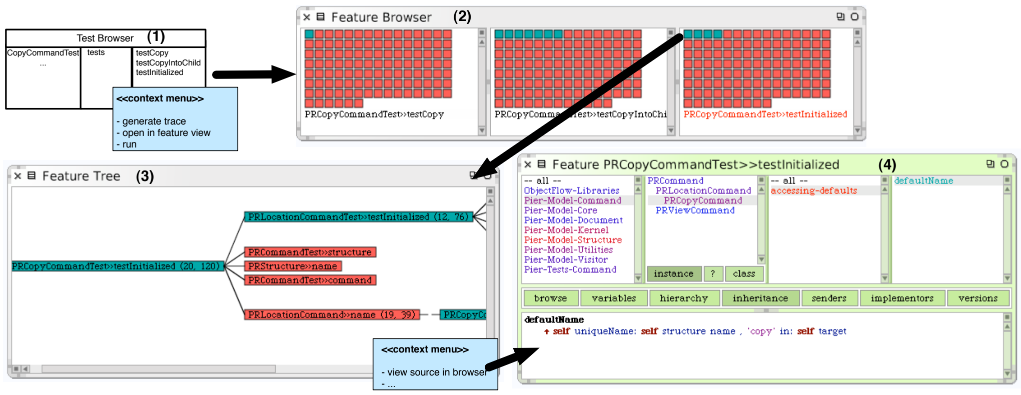
To tackle the shortcoming of the existing Squeak IDE not having any explicit representation of a system's dynamics, we propose to enhance the static perspective with a feature perspective of a software system. We present a novel feature-centric environment providing support for visual representation, interactive exploration and navigation of a system's features [1]. Furthermore, we enrich the source code with information about its execution, *i.e.,* run-time types of variables or precise information about methods being invoked from within a given method [2]. A third enhancement enables the developer to query gathered dynamic information from within the IDE, for instance to reveal what methods get invoked most frequently during the execution of a software feature [3].

In the following, we give a short overview of how these three means to integrate dynamic information are implemented in our Hermion IDE, an enhanced development environment for Squeak based on the OmniBrowser framework [4].

2 Feature-centric Environment

We embed the feature-centric environment tightly in the traditional Smalltalk IDE to augment this IDE with a feature perspective of a software. This feature environment complements the traditional structural and purely textual representation of source code in a browser by presenting the developer with interactive, navigable visualizations of features in three distinct but complementary views. These views are enriched with metrics to provide the software engineer with additional information about the relevancy of source artifacts (*i.e.*, classes and methods) to specific features.

A schema of the feature-centric environment is shown in Figure 1. (1) is the test runner which is not directly part of the feature-centric environment but a separated tool. Tests are recorded scripts to execute features, but developers can also directly run the system like end-users to trigger the execution of features. The feature-centric environment contributes three different visualizations for one and the same feature: (2) the compact feature overview, (3) the feature tree view, and (4) the feature artifact browser.



```
defaultName
    ↑ self uniqueName: self structure name , 'copy' in: self target
```

Fig. 1. The Elements of our Feature Browser Environment

Compact Feature Overview. The Compact Feature Overview presents a visualization of two or more features represented in a compacted form. This view represents a feature as a collection of all methods used in the feature as a result of capturing its execution trace. Each method is displayed as a small colored box; the color represents the *Feature Affinity* value. Cyan hereby represents a method only used in this particular feature in the list of features, while red on the other hand means that this method is used by all features in the list. The methods are sorted according to their *Feature Affinity* value. The software engineer decides how many features she wants to visualize at the same time (see Figure 1 (2)). Clicking on a method box in the Compact Feature View opens the Feature Tree View, which depicts a call tree of the execution trace. This visualization reveals the method names and order of execution.

Feature Tree. This view presents the method call tree, captured as a result of exercising one feature (see Figure 1 (3)). The first method executed for a feature (*e.g.,* the "main" method) forms the root of this tree. Methods invoked in this root node form the first level of the tree, hence the nodes represent methods and the edges are message sends from a sender to a receiver. The nodes of the tree are again colored according to their *Feature Affinity* value.

Feature Artifact Browser. The source artifacts of an individual feature are presented as text in the *feature artifact browser* (see Figure 1 (4)). It exclusively displays the classes and methods actually used in the feature. This makes it much easier for the user to focus on a single feature of the software. The*feature artifact browser* is an adapted version of the standard class browser available in Squeak. However, this version of the class browser not only presents static source artifacts, but also the feature affinity metric values by coloring the names of classes and methods accordingly.

The three distinct visualizations provided by the feature browser are tightly interconnected so that developers do not lose the overview when performing a maintenance task. For instance, the user selects a method in the compact feature overview, the tree opens with all occurrences of the selected method. In the tree the developer chooses to view a method in the feature artifact browser.

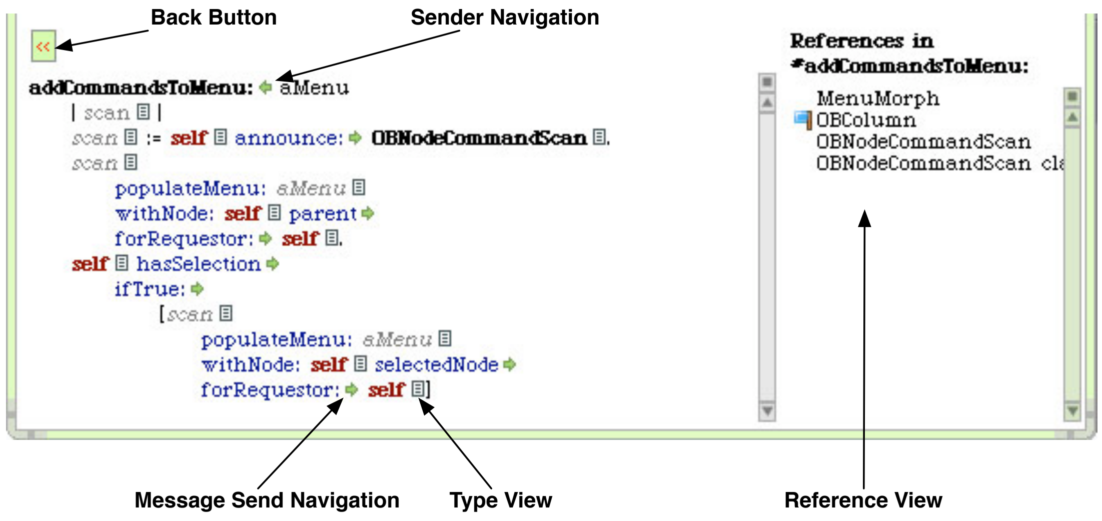
3 Enriching Source Code with Dynamic Information

Besides integrating a feature representation in the IDE, we also enrich directly the view on static source artifacts, in particular the source code view, with dynamic information. In Squeak it is difficult to get a complete understanding for the running of a method by just reading the source code as there is no explicit type information available. Without this information, we often do not know what types variables get at run-time and what methods get invoked, especially when polymorphism is applied. We mitigate this problem by analyzing the running of methods to give developers insights into the dynamics directly when they are working with the source code.

We enrich the static source code perspective with three novel enhancements: (i) run-time types for variables, (ii) precise message send navigation, and (iii) dynamic references occurring in selected classes or methods. These enhancements are available for entities (*e.g.,* packages or classes) the developer has chosen to observe dynamically. As soon as the application using these entities gets executed, Hermion takes care of collecting run-time information to immediately enhance the source code view. Figure 2 summarizes these enhancements.

Run-time Types of Variables. All variables that have been accessed at run-time get an icon next to them in the source code view. Clicking on this icon shows the run-time types this variable got and how often it has been accessed for each type during the system's execution. The developer can browse to the class corresponding to each type in the list by choosing the appropriate list item.

Message Send Navigation. For message send we also add an icon next to them in the source code view. If it is a polymorphic message send, clicking on the icon brings up a list with all methods being invoked, including their respective receiver types. If there is only one receiver, the developer directly navigates to the method that has been invoked at runtime at that place in source code.
Reference View. The reference view shows all classes that have been referenced at run-time in the selected class or method. The developer can navigate these referenced classes or see where in source code the reference was done.



```
addCommandsToMenu: aMenu
    | scan |
    scan := self announce: OBNodeCommandScan.
    scan
        populateMenu: aMenu
        withNode: self parent
        forRequestor: self.
    self hasSelection
        ifTrue:
            [scan
                populateMenu: aMenu
                withNode: self selectedNode
                forRequestor: self]
```

```
References in
#addCommandsToMenu:
  MenuMorph
  OBColumn
  OBNodeCommandScan
  OBNodeCommandScan cl
```

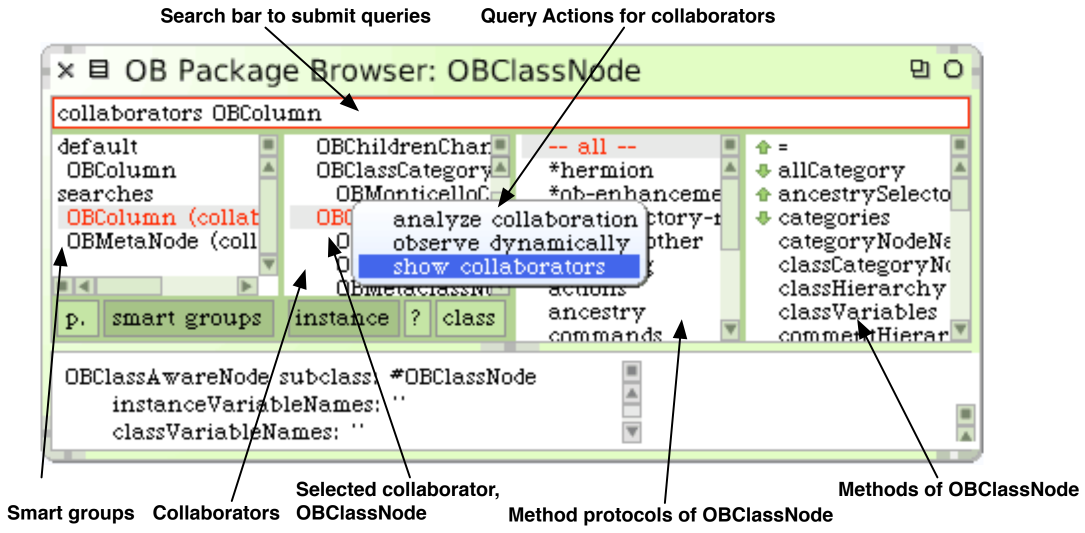
Fig. 2. Enriching source code with dynamic information

4 Querying Dynamic Information

Finally, we add a search bar on the top of the class browser of Squeak to enable the developer to enter textual search queries to retrieve dynamic information. Such a query is for instance *SHOW collaborators OF OBColumn* to get all classes collaborating with *OBColumn* at run-time. The result is then permanently stored in a so-called smart group accessible from within the IDE, as illustrated in Figure 3. Query results can be analyzed further, for instance to study how a class is dynamically collaborating with *OBColumn*.

5 Conclusion

In this paper we presented an integration of dynamic information into the Squeak IDE by (i) explicitly representing features, (ii) embedding run-time information



```
OBClassAwareNode subclass: #OBClassNode
    instanceVariableNames: ''
    classVariableNames: ''
```

Fig. 3. Querying dynamic information, results are accessible in smart groups

into the source code, and (iii) providing query facilities covering dynamic information. An IDE augmented with dynamic information is superior to an environment purely focusing on static source artifacts, as these enhancements eventually enhance software understanding for the developer and ease maintenances activities as shown in [1] or [2]. In particular for dynamic languages such as Smalltalk it is crucial to have insights into the running of a system (*e.g.,* revealing run-time type information) while working with static source entities.

References

1. D. Röthlisberger, O. Greevy, O. Nierstrasz, Feature driven browsing, in: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 79–100.
2. D. Röthlisberger, O. Greevy, O. Nierstrasz, Exploiting runtime information in the ide, in: Proceedings of the 2008 International Conference on Program Comprehension (ICPC 2008), 2008.
3. D. Röthlisberger, Querying runtime information in the ide, in: Proceedings of the 2008 workshop on Query Technologies and Applications for Program Comprehension (QTAPC 2008), 2008.
4. A. Bergel, S. Ducasse, C. Putney, R. Wuyts, Creating sophisticated development tools with OmniBrowser, Journal of Computer Languages, Systems and Structures 34 (2-3) (2008) 109–129.