Finding the Missing Eclipse Perspective: the Runtime Perspective

Andrew Giang Marwan Abi-Antoun
Department of Computer Science, Wayne State University, Detroit, Michigan, USA
{andrewgiang, mabiantoun}@wayne.edu

Abstract

When evolving object-oriented code, developers need to understand both the code structure and the runtime structure. However, Integrated Development Environments such as Eclipse still predominantly emphasize the code structure. We propose to add to Eclipse a *Runtime Perspective* to complement the existing perspectives such as the Java *code* perspective and the *debugging* perspective, and to present to developers a hierarchical abstract runtime structure in terms of abstract objects and relations between them.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming

Keywords Object structure; Object diagrams

1. Introduction

Today's Integrated Development Environments (IDEs) still do *not* present to *object*-oriented developers a view of the *objects* in the system! Instead, IDEs seem to emphasize a *class*- oriented view of the system at design-time. This is particularly challenging for beginner programmers while they learn object-oriented design patterns and frameworks.

For example, the popular Eclipse IDE presents to developers a *hierarchy of classes* where classes are organized by packages, as in the Package Explorer. Eclipse also shows a *type hierarchy* that shows the inheritance hierarchy of the selected classes and interfaces. If developers want to understand the objects in a system, they have to either mentally visualize them, or run the system and use the *debug* perspective in Eclipse and wade through a heap of *concrete objects* in the debugger, examining *specific instances*. There are often too many specific instances. For some coding tasks that require knowing how many instances of a class are created, a debugger is crucial. But for many program comprehension tasks, *specific* instances may not matter.

In previous work, Abi-Antoun and Aldrich developed a static analysis to extract from a program with annotations an approximation of the runtime structure as a global, hierarchical, Ownership Object Graph (OOG) [1]. A controlled experiment found that developers who have access to a hierarchy of *abstract objects* make code modifications faster and browse less irrelevant code [2]. The previous tool, however, was not well integrated with the Eclipse IDE and lacked several features. In this work, we add to Eclipse a novel, *Runtime Perspective*, to complement the current perspectives.

2. High-Level Description

The Runtime Perspective (Fig. 1) is integrated with various features in Eclipse such as the Javadoc viewer and the Package Explorer and has several views that are based on information from the abstract runtime structure. The abstract runtime structure is extracted by another tool and stored in a file. The Runtime Perspective loads the file and uses the information to populate the various views. In the abstract object hierarchy, the objects that are architecturally relevant are higher in the hierarchy than low-level implementation details such as data structures.

The **Abstract Object Tree (1)** displays a hierarchy of abstract objects. From a selected abstract object, the **Abstract Stack (2)** shows the nested abstract interpretation contexts that lead to the creation of the selected abstract object. Ultimately, each abstract object is due to an object creation expression in the code. Several creation expressions, scattered across several Java classes, may be represented by the same abstract object. Moreover, one object creation expression may create several abstract objects that are in different contexts. Furthermore, the Abstract Stack View has a traceability feature: when the developer double-clicks on an entry, the tool selects the corresponding lines of code. This is similar to the stack in the debug perspective.

The **Summary View (3)** shows the list of the Most Important Classes (MICs), the Most Important Related Classes to a class (MIRCs), and the Most Important Methods of a class (MIMs). The Summary View suggests the MICs for the open project. When a developer picks a class `C` from the MICs, the Summary View automatically opens the class in

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514575

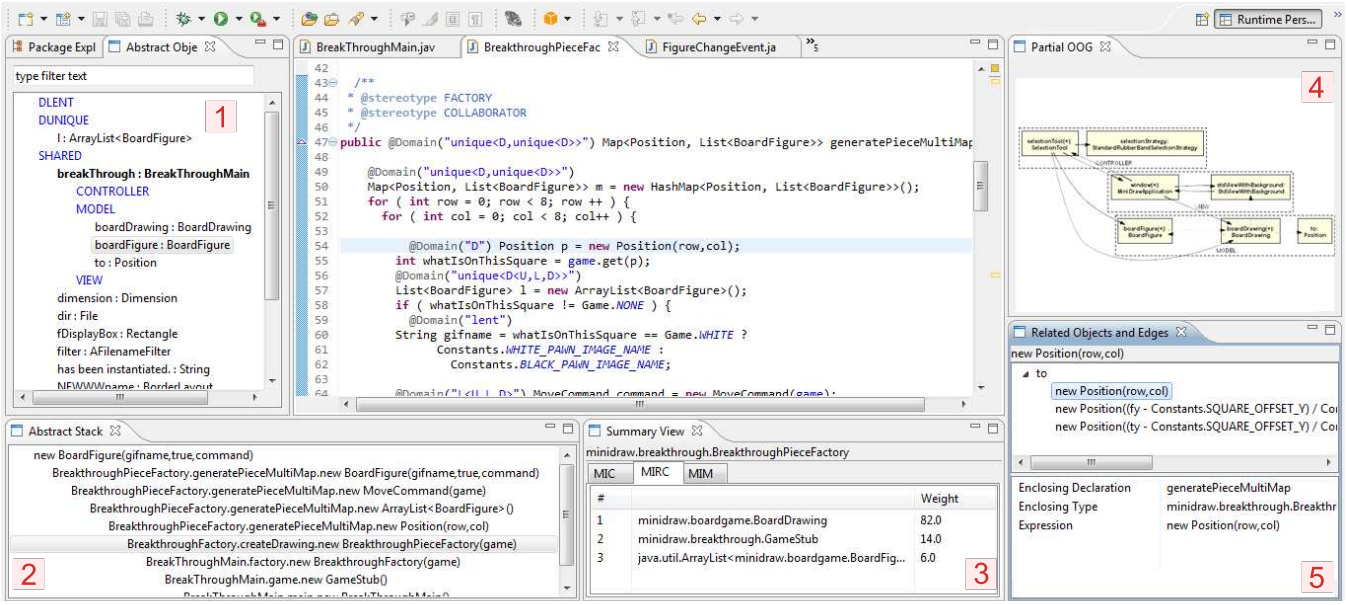

Figure 1. The Eclipse *Runtime Perspective*.

the Eclipse Java editor. As the developer navigates to a class `C` in the project or using the Package Explorer, the tool automatically displays the list of MIRCs or MIMs related to `C`. While browsing classes and methods in the IDE, when a developer selects a field declaration that is of an interface, the Runtime Perspective automatically shows the Most Important Classes behind an Interface (MCBIs) as a Quick Fix in Eclipse. The **Graph View (4)** displays a partial OOG that shows only the nodes and edges OOG that are related to the class that the developer is working on.

If the developer selects some code elements in the Java editor, the **Related Objects/Edges View (5)** displays a list of related objects and edges that the selected code element maps to in the abstract runtime structure. The developer can expand each related object or edge and go to the corresponding lines of code in the Eclipse project.

3. Description of the Summary View

The demonstration will illustrate how the Summary View works using a small system, MiniDraw, a pedagogical object-oriented framework. In the demonstration, the audience will see how a developer completes a code modification task. The task given to the developer is to validate the movement of a piece on a boardgame [2].

The Summary View lifts the information about architectural relevance back to the types. For example, the Summary View considers the MICs to be the classes of the objects at the top level of the hierarchy. All the top-level objects are included in the MICs, then ranked based on the number of filtered edges. To find the MIRCs for a given class C, the strategy traverses the set of filtered edges and collects the union of edges that have a source or a destination that is of type C or a subtype of C. To find the MIMs for a given class C, the Summary View ranks the methods by the number of edges they create. To find the MCBIs, for a given interface, the Summary View finds all the classes that implement the interface, then ranks these classes using a similar strategy.

One Summary View strategy uses the number of incoming, outgoing, and self edges from an abstract object to rank the corresponding classes and methods. Using the MICs tab in the Summary View, the developer determines which of the most important classes may implement movement or drawing items to the board. Since the Summary View shows a ranked subset of classes, it is easier for a developer to decide which classes are important and where to begin. From the list of MICs, there are two classes that stand out for our task: `BoardDrawing` and `BoardFigure`. From here, the developer can start to look at the MIRCs and the MIMs of these classes. The search ends when the developer finds the method `move` in `GameStub`, the first method listed in the MIMs of `GameStub`.

When encountering a field declaration that is of an interface type, the developer typically uses the Eclipse Type Hierarchy to find all possible classes behind the interface. This list can be potentially long, and does not rank these classes in any way except using the depth of the inheritance tree. Since they are based on the abstract runtime structure, the MCBIs in the Runtime Perspective are often a smaller, more precise set of possible concrete types.

References

[1] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *OOPSLA*, pages 321–340, 2009.

[2] N. Ammar and M. Abi-Antoun. Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks. In *WCRE*, pages 367–376, 2012.