The Eclipse Runtime Perspective for Object-Oriented Code Exploration and Program Comprehension

Marwan Abi-Antoun Andrew Giang Sumukhi Chandrashekar Ebrahim Khalaj

Department of Computer Science, Wayne State University

{mabiantoun, andrewgiang, sumukhic, mekhalaj}@wayne.edu

Abstract

We propose a novel Eclipse Perspective, the Runtime Perspective, that makes a global hierarchy of abstract objects a first-class view of an object-oriented system at design-time. The perspective includes many views that complement the existing views in the Java Development perspective: an Abstract Object Tree with a search feature to complement the Package Explorer, an Abstract Stack to complement the Call Hierarchy, and a Partial Object Graph, to complement the class diagrams extracted by many existing plugins.

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

Keywords Object graphs; Object diagrams

1. Introduction

Software maintenance costs around 50%–90% of the total costs over the lifecycle of a software system, out of which 50% is spent in program comprehension [4]. To aid comprehension, different views have been proposed. Many focus on the static code structure [12], others focus on the runtime structure, also called the dynamic or execution structure [8]. Yet, some others focus on deployment.

In object-oriented design, developers need to understand both the code structure in terms of classes and relationships between them such as inheritance, as well as the runtime structure in terms of objects and relations among them [9]. Still, today's Integrated Development Environments (IDEs) present to developers mainly the code structure of a system. As an example, the popular Eclipse IDE presents to object-oriented Java developers a predominantly class-oriented view of the system. In this paper, we call a *hierarchy of classes* any tree-based representation of a project in terms of its packages, any nested packages, classes inside packages, and inner classes nested inside classes, as a hierarchy of classes, exemplified by the Package Explorer or the Outline View in Eclipse. This hierarchy is not to be confused with a *type hierarchy* that shows interfaces and classes, and any classes implementing them or extending from them, which is also supported by Eclipse, and invoked on demand.

From looking at the code, using a hierarchy of classes, or using a type hierarchy, it is hard to understand the runtime structure, i.e., what objects are created at runtime from these classes. While there are many tools that support understanding the code structure [12], the tools to support understanding the runtime structure are fewer and less mature.

Moreover, it is not straightforward to understand the runtime structure from the code structure. For some systems, the runtime structure is very different from the code structure [2]. In particular, some object-oriented design patterns such as Observer or coding idioms such as programming to interfaces lead to greater differences. Today, to understand the object structure, developers typically launch a debugger and step through the executing code. However, the debugger requires that developers completely switch their mental model and wade through a heap of concrete objects.

For some debugging tasks that involve knowing how many instances of a class are created, or how one instance is related to how many other instances, a debugger is crucial. But for many program comprehension tasks, understanding the many specific instances or *concrete* objects may not matter. Instead, it may be useful to abstract objects and merge conceptually similar objects or those that play the same role into one *abstract object*.

To present to developers a manageable number of abstract objects, we extract a *hierarchy of abstract objects* in the form of a global, hierarchical object graph, the Ownership Object Graph (OOG) [1]. In the OOG, one abstract object represents zero or more runtime objects, and one abstract edge represents a relation between the corresponding objects. Moreover, the OOG uses *abstraction by hierarchy*: architecturally significant objects are near the top of hierarchy and imple-

ETX '14, October 21, 2014, Portland, OR, USA..
Copyright © 2014 ACM 978-1-4503-2530-1/14/10…$15.00.
http://dx.doi.org/10.1145/10.1145/2688130.268813

mentation details such as data structures are further down. The hierarchy enables developers to achieve both a high-level and a detailed understanding of the system [17].

The OOG is extracted using static analysis. As a result, it can be used at design time without running the system. Unlike a typical IDE that shows the classes of the objects at design time, or a debugger that shows the concrete objects obtained at runtime, the OOG approximates the runtime structure and presents abstract objects at design time, and bridges the gap between the static code structure and the runtime structure.

Contributions. This paper contributes a novel Eclipse development-time perspective that mines information from the OOG and presents that information to developers using various diagrammatic and non-diagrammatic views.

Outline. The rest of this paper is structured as follows. Section 2 gives some background on the OOG that is used to populate the Runtime Perspective. Section 3 presents the Runtime Perspective by example. Section 4 presents some applications of the Runtime Perspective then discusses some limitations and future work. Next, we discuss related work (Section 5) and conclude.

2. Background

This section briefly discusses the process involved in extracting the OOG [1, 18]. To extract an OOG that is hierarchical and conveys design intent, we place each object into a *domain*, a named, conceptual group of objects that expresses design intent. This information is missing in plain Java code, so we supply it with annotations using available language support. The annotations are currently added manually, but are amenable in principle to type inference. Runtime objects from the heap are then abstracted to pairs to types and domains. That is, one or more runtime objects of the same type and in the same domain are considered to have the same role, and are abstracted into one abstract object. In particular, any abstract object can also have one or more nested domains that contain other abstract objects. This representation achieves a hierarchy of abstract objects and can distinguish an abstract object by its type, its domain, and its position in the object hierarchy. The process of adding annotations is guided by the principle of abstraction by hierarchy.

In the OOG, a relationship between two abstract objects is represented by an abstract edge. For example, a field reference leads to a points-to edge between the corresponding abstract objects. The static analysis also extracts dataflow communication between objects, which represents usage relations due to field reads, field writes, or method invocations.

Graphically, a domain is represented by a dashed-line box with the domain name capitalized. An abstract object is represented by a yellow- (gray in black-and-white) colored solid box labeled `a:A` where `a` is a display name and `A` is the declared type in the code. A thin edge between abstract objects denotes a points-to relation, a thick edge denotes a dataflow relation.

Next, we briefly discuss the process that developers follow to extract the OOG. To start, developers add annotations to the code, expressing their design intent. A type checker checks that the annotations are consistent with each other and with the code. Once the annotations are type checked, a static analysis extracts the OOG based on the code and the type checked annotations. The OOG is saved into an external file. In the next section, we discuss the Runtime Perspective, which loads the OOG from an external file and queries it.

3. Runtime Perspective

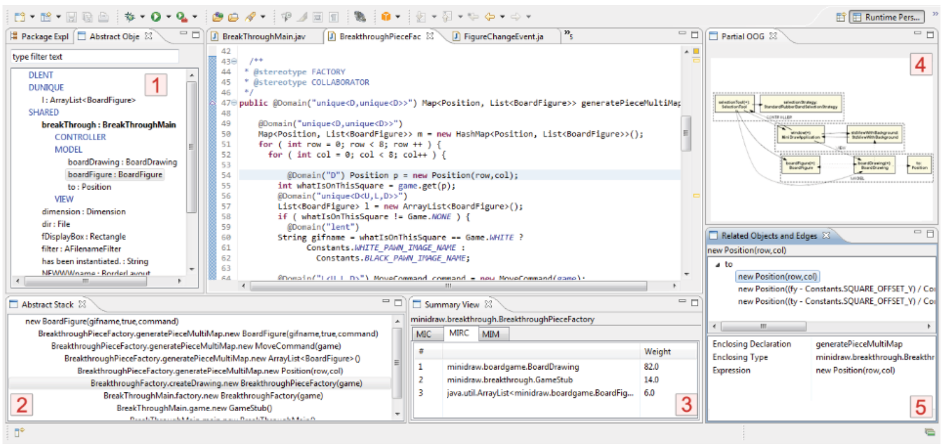
While using the Eclipse Java development perspective, and without running the program, developers can switch to the Runtime Perspective (Fig. 1), which loads several views that display information based on the OOG around the Java editor, and complement the existing views such as the Eclipse Package Explorer. In this section, we use as a running example a board game called Breakthrough, which is built on a small, pedagogical, object-oriented framework called MiniDraw [7, 14].

This section discusses each of the following views of the Runtime Perspective and contrasts it with its closest counterpart in one of the standard Eclipse perspectives:

- Eclipse Package Explorer vs. Abstract Object Tree (numbered 1 in Fig. 1) in the Runtime Perspective;
- Eclipse Call Hierarchy vs. Abstract Stack (numbered 2) in the Runtime Perspective;
- Eclipse Java Search vs. Related Objects and Edges (numbered 5) in the Runtime Perspective;
- Eclipse File Search vs. Object Search in the Runtime Perspective; and
- Tool-extracted UML class diagram vs. Partial OOG (numbered 4) in the Runtime Perspective.

Eclipse Package Explorer. Fig. 2(a) shows a hierarchy of classes, organized by packages, typically sorted alphabetically. The substructure of a class consists of field declarations, method declarations, and inner classes. The example below shows three packages, `minidraw`, `minidraw.boardgame` and `minidraw.breakthrough`. Clicking on `minidraw.breakthrough` shows all the classes in the package for example, `BreakthroughFactory`, `BreakThroughMain` and so on. Furthermore, the substructure of the class `BreakThroughMain` consists of two method declarations: `main(String)` and `init()`, several fields: `factory` of type `BreakthroughFactory`, `game` of type `GameStub` and so on. From a selected element, the developer can go to the corresponding declaration.

Abstract Object Tree. Fig. 2(b), in contrast, shows a hierarchy of abstract objects and domains queried from the OOG. The developers can search for abstract objects by name or by type in the object tree.



type filter text

- DLENT
- DUNIQUE
  - l : ArrayList<BoardFigure>
- SHARED
  - breakThrough : BreakThroughMain
    - CONTROLLER
    - MODEL
      - boardDrawing : BoardDrawing
      - boardFigure : BoardFigure
      - to : Position
    - VIEW
  - dimension : Dimension
  - dir : File
  - fDisplayBox : Rectangle
  - filter : AFilenameFilter
  - has been instantiated. : String

```
 /**
 * @stereotype FACTORY
 * @stereotype COLLABORATOR
 */
public @Domain("unique<D,unique<D>>") Map<Position, List<BoardFigure>> generatePieceMultiMap

   @Domain("unique<D,unique<D>>")
   Map<Position, List<BoardFigure>> m = new HashMap<Position, List<BoardFigure>>();
   for ( int row = 0; row < 8; row ++ ) {
     for ( int col = 0; col < 8; col++ ) {

       @Domain("D") Position p = new Position(row,col);
       int whatIsOnThisSquare = game.get(p);
       @Domain("unique<D<U,L,D>>")
       List<BoardFigure> l = new ArrayList<BoardFigure>();
       if ( whatIsOnThisSquare != Game.NONE ) {
         @Domain("lent")
       String gifname = whatIsOnThisSquare == Game.WHITE ?
             Constants.WHITE_PAWN_IMAGE_NAME :
               Constants.BLACK_PAWN_IMAGE_NAME;
```

Abstract Stack

```
new BoardFigure(gifname,true,command)
  BreakthroughPieceFactory.generatePieceMultiMap.new BoardFigure(gifname,true,command)
    BreakthroughPieceFactory.generatePieceMultiMap.new MoveCommand(game)
      BreakthroughPieceFactory.generatePieceMultiMap.new ArrayList<BoardFigure>()
        BreakthroughPieceFactory.generatePieceMultiMap.new Position(row,col)
          BreakthroughFactory.createDrawing.new BreakthroughPieceFactory(game)
            BreakThroughMain.factory.new BreakthroughFactory(game)
              BreakThroughMain.game.new GameStub()
```

Summary View

minidraw.breakthrough.BreakthroughPieceFactory

MIC | MIRC | MIM

#		Weight
1	minidraw.boardgame.BoardDrawing	82.0
2	minidraw.breakthrough.GameStub	14.0
3	java.util.ArrayList<minidraw.boardgame.BoardFig...	6.0

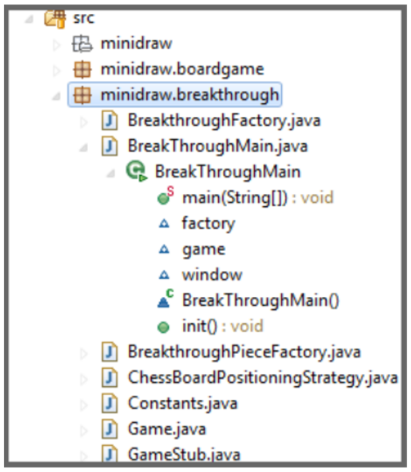
Related Objects and Edges

new Position(row,col)

- to
  - new Position(row,col)
  - new Position((fy - Constants.SQUARE_OFFSET_Y) / Co
  - new Position((ty - Constants.SQUARE_OFFSET_Y) / Co

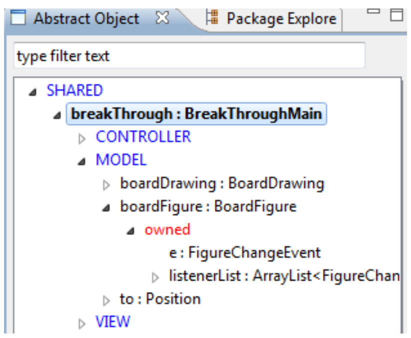
Enclosing Declaration	generatePieceMultiMap
Enclosing Type	minidraw.breakthrough.Breakthr
Expression	new Position(row,col)

Figure 1. Runtime Perspective. The Java editor is in the center. Some views are numbered.



- src
  - minidraw
  - minidraw.boardgame
  - minidraw.breakthrough
    - BreakthroughFactory.java
    - BreakThroughMain.java
      - BreakThroughMain
        - main(String[]) : void
        - factory
        - game
        - window
        - BreakThroughMain()
        - init() : void
    - BreakthroughPieceFactory.java
    - ChessBoardPositioningStrategy.java
    - Constants.java
    - Game.java
    - GameStub.java

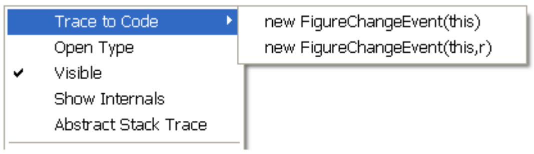
(a) Eclipse Package Explorer.



type filter text

- SHARED
  - breakThrough : BreakThroughMain
    - CONTROLLER
    - MODEL
      - boardDrawing : BoardDrawing
      - boardFigure : BoardFigure
        - owned
          - e : FigureChangeEvent
          - listenerList : ArrayList<FigureChan
      - to : Position
    - VIEW

(b) Abstract Object Tree.



(c) Abstract Object Tree context-menu.

Figure 2. Package Explorer vs. Abstract Object Tree.

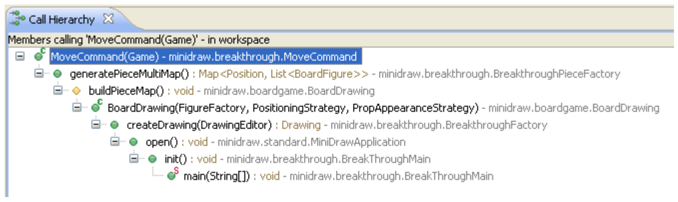
The example below indicates that the abstract object `breakThrough` of type `BreakThroughMain` declares three domains: `CONTROLLER`, `MODEL`, and `VIEW`. The domain `MODEL` consists of abstract objects such as `boardDrawing` of type `BoardDrawing` and `boardFigure` of the type `BoardFigure`. The abstract object `boardFigure` has a domain owned that has another abstract object `listenerList` of type `ArrayList<FigureChangeEvent>`. This forms a hierarchy of abstract objects. When the developers trace to code, they trace to expressions associated with the abstract objects rather than declarations. For example, the abstract object `listenerList` traces to an object creation expression in the class `AbstractFigure`. In general, when the developers trace to code from an abstract object, they can go to one or more object creation expressions. From an abstract edge, they can trace to one or more field declaration, or field read, field write, or method invocation.

In the Abstract Object Tree, the context-menu (Fig. 2(c)) has several features:

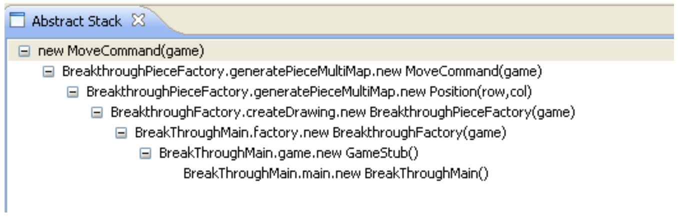
- Trace to Code: traces back to one or more object creation expressions that correspond to this abstract object;
- Open Type: opens the type declaration `C` for an abstract object `o:C`; the developer can optionally invoke the Eclipse Type Hierarchy on that type;
- Abstract Stack Trace: opens the stack trace that shows the nested abstract interpretation contexts that lead to the creation of this abstract object;
- Visible: toggles whether this object should be displayed in the Partial OOG. Even if not visible in the Partial OOG, it is still visible in the tree.

Eclipse Call Hierarchy. Fig. 3(a) shows the caller and callees for a selected method. In the example, the constructor for `MoveCommand` is called by the method `generatePieceMultiMap`, which is called by the method `buildPieceMap`, which is called from the constructor of `BoardDrawing`, and so on.

Abstract Stack. The Abstract Stack (Fig. 4) shows the nested abstract interpretation contexts that led to the creation of an abstract object. For example, the analysis an-


```
Call Hierarchy
Members calling 'MoveCommand(Game)' - in workspace
MoveCommand(Game) - minidraw.breakthrough.MoveCommand
  generatePieceMultiMap() : Map<Position, List<BoardFigure>> - minidraw.breakthrough.BreakthroughPieceFactory
    buildPieceMap() : void - minidraw.boardgame.BoardDrawing
      BoardDrawing(FigureFactory, PositioningStrategy, PropAppearanceStrategy) - minidraw.boardgame.BoardDrawing
        createDrawing(DrawingEditor) : Drawing - minidraw.breakthrough.BreakthroughFactory
          open() : void - minidraw.standard.MiniDrawApplication
            init() : void - minidraw.breakthrough.BreakThroughMain
              main(String[]) : void - minidraw.breakthrough.BreakThroughMain
```

(a) Eclipse Call Hierarchy.


```
Abstract Stack
new MoveCommand(game)
  BreakthroughPieceFactory.generatePieceMultiMap.new MoveCommand(game)
    BreakthroughPieceFactory.generatePieceMultiMap.new Position(row,col)
      BreakthroughFactory.createDrawing.new BreakthroughPieceFactory(game)
        BreakThroughMain.factory.new BreakthroughFactory(game)
          BreakThroughMain.game.new GameStub()
            BreakThroughMain.main.new BreakThroughMain()
```

(b) Abstract Stack.

Figure 3. Call Hierarchy vs. Abstract Stack.

```
A a = new A();
a.mA();
class A {
    void mA() {
        B b = new B();
    }
}
class B {
    void mB() {
        C c = new C();
    }
}
```

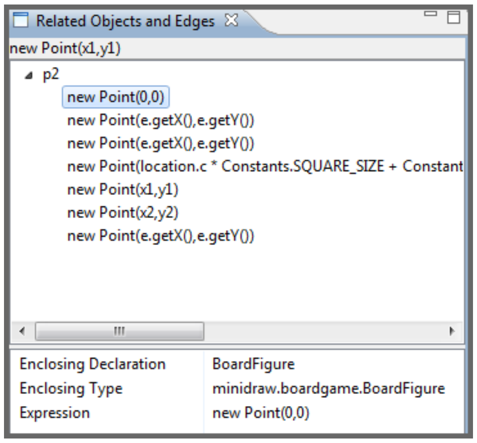
Abstract Stack for the code above

```
new C()
  +- B.mB() [new C()]
    +- A.mA() [new B()]
      +- Main.main() [new A()]
```

Figure 4. Abstract Stack example.

alyzes the `new A()` expression, which leads to analyzing all the method declarations in class `A`. One of them has a `new B()` expression, which in turn leads to analyzing the method declarations in class `B`, then the method declarations in class `C`. In program analysis terms, this view exposes to developers the notion of *object sensitivity*. For MiniDraw (Fig. 3(b)), the abstract object `boardFigure` of the type `BoardFigure` is due to an object creation expression in the method `generatePieceMultiMap()` in `BreakthroughPieceFactory`.

Related Objects and Edges. Fig. 5 shows related expressions in the application that are associated with the selected line of code. It identifies all the code elements that map to the same abstract object or abstract edge in the OOG. The example below shows all abstract objects that are related to the currently selected AST node, p2. When each of these related expressions are selected, information such as the enclosing type of the declaration, the enclosing declaration are retrieved.


```
Related Objects and Edges
new Point(x1,y1)
  p2
    new Point(0,0)
    new Point(e.getX(),e.getY())
    new Point(e.getX(),e.getY())
    new Point(location.c * Constants.SQUARE_SIZE + Constant
    new Point(x1,y1)
    new Point(x2,y2)
    new Point(e.getX(),e.getY())
```

Enclosing Declaration	BoardFigure
Enclosing Type	minidraw.boardgame.BoardFigure
Expression	new Point(0,0)

Figure 5. Related Objects and Edges.

Object Search is used to find all the abstract objects of a given type and the associated edges. Fig. 6 shows the results of invoking Object Search when the class `NullTool` is loaded in the Java editor using a context menu item called Find Abstract Objects. The view shows there are two abstract objects for the same class `NullTool`. The abstract object `fTool: NullTool` traces to an object creation expression in `MiniDrawApplication`. The views also shows a points-to edge from the abstract object `windows: MiniDrawApplication` to this abstract object.

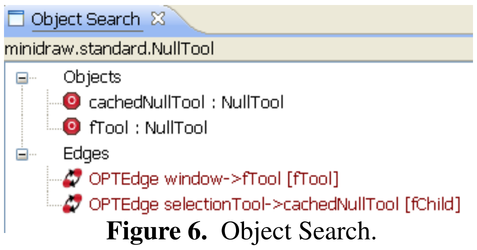

```
Object Search
minidraw.standard.NullTool
  Objects
    cachedNullTool : NullTool
    fTool : NullTool
  Edges
    OPTEdge window->fTool [fTool]
    OPTEdge selectionTool->cachedNullTool [fChild]
```

Figure 6. Object Search.

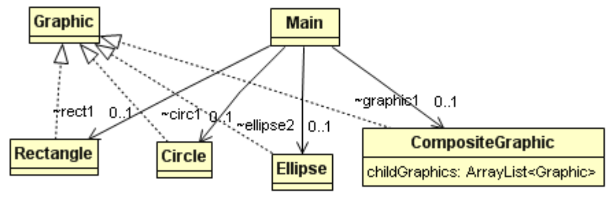
4. Applications

We anticipate several applications for the Runtime Perspective, mainly in an educational setting.

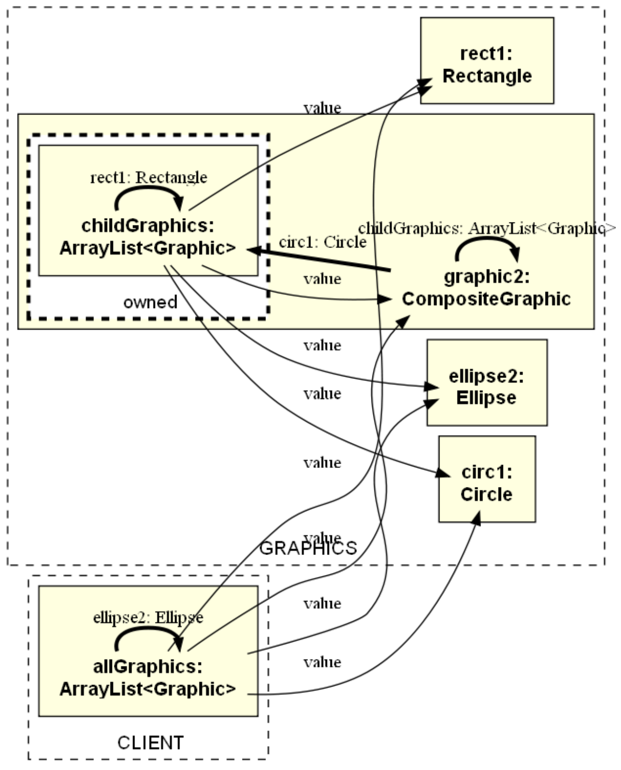
4.1 Explaining Structural Design Patterns

The Design Patterns book [9] uses both class and object diagrams to explain several structural design patterns such as Proxy, Mediator and Composite. We used an Eclipse plugin, ObjectAid [15], to extract a class diagram for a few classes involved in a Composite design pattern (Fig. 7(a)).

Using the Runtime Perspective, the developer can display a Partial OOG for objects of those same classes (Fig. 7(b)). The OOG makes the pointer structures more explicit. For example, there is a cycle in the OOG when going from `graphic2:CompositeGraphics` to `childGraphics` then back to `graphic2`. The OOG, however, also suffers from some imprecision. For example, it shows that both



(a) Class diagram.



(b) OOG.

Figure 7. Composite pattern: class diagram vs. OOG.

allGraphics and childGraphics refer to all abstract objects of type Graphic such as Rectangle, Circle, etc. Some of these edges may be false positives and may not exist at runtime. Indeed, childGraphics, which references the children of a composite, is likely to refer to fewer objects than allGraphics, which references all the graphical shapes in the system.

4.2 Explaining Shallow vs. Deep Object Copying

Based on our experience teaching undergraduates, novice object-oriented programmers seem to struggle with the notion of deep vs. shallow object copying, copy constructors, or the notion of value equality. In an educational setting, one can use an OOG to explain the difference more clearly than looking at the code (Fig. 9). For example, the diagram makes it more explicit than the contents of the buff array are being copied from one object to another. The code that illustrates object copying is shown with the annotations (Fig. 8).

```
@Domains({ "DOM1", "DOM2" })
class Main {
  Main() {
    @Domain("DOM1")Text o1 = new Text("t1");
    @Domain("DOM2")Text o2 = new Text("t2");
    if (!o1.equals(o2))
      System.out.println("Different contents!");
  }
}
@Domains({"owned"})
class Text {
  @Domain("owned") char[] buff;
  int size = -1;

  Text(@Domain("shared") String text) {
    int textsize = text.length();
    buff = new char[textsize];
    size = textsize;
    for (int i = 0; i < textsize; i++)
      this.buff[i] = text.charAt(i);
  }
  boolean equals(@Domain("lent") Text o) { ... }
}
```

Figure 8. Code for object copying, with annotations.

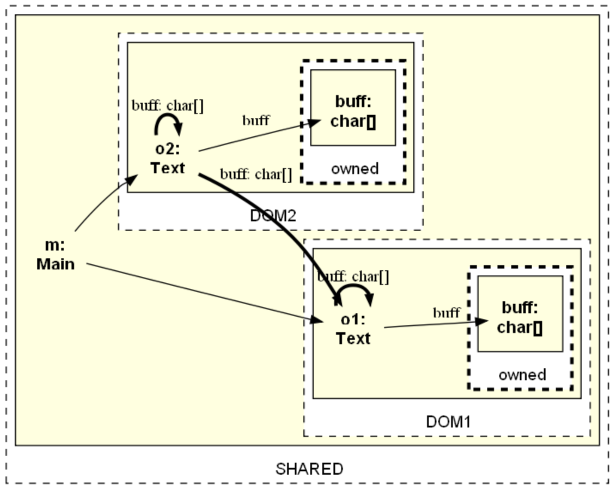


Figure 9. OOG illustrating object copying.

4.3 Limitations and Future Work

Stale OOGs. The Runtime Perspective is primarily a code exploration tool. As developers make changes to the code, the changes may add more objects or relations between new or existing objects. Thus, the OOG may become stale and no longer reflect those new objects or relations. To remedy that, the developers must update the annotations, re-extract the OOG and re-load the Runtime Perspective.

Future Work. We plan to evaluate the Runtime Perspective in user studies. Our previous controlled experiment [3] showed that developers performing coding tasks benefited from having access to OOGs as diagrams of the runtime structure. The participants, however, used an older version of the tools for the OOG that had one monolithic view, over-emphasized the OOG visualization and lacked the other views that present information about the abstract runtime structure in non-diagrammatic ways. We also plan to use the Runtime Perspective in an educational setting, particularly in undergraduate software design courses.

5. Related Work

There is a large body of research on software visualization and code exploration. We briefly mention some of the tools that are most directly related.

Objektgraph. The closest tool in spirit is Objektgraph [6], which shows flat object diagrams rather than hierarchical abstract object graphs. In some sense, object diagrams are closer depictions of the runtime structure and show multiple instances of the same type without any abstraction. But since these object diagrams are neither abstract nor hierarchical, they neither scale well nor convey high-level understanding.

Exploration of the code structure. There are many tools that focus on exploring the code structure [12], focusing on class diagrams and other views of the static code structure such as module views showing packages.

Call graphs. Many tools focus on visualizing call graphs [5]. The Runtime Perspective takes an object-centric view, showing abstract objects first then showing dataflow edges to capture method invocations.

Heap exploration. HeapViz [10] enables developers to visualize and interactively explore snapshots of the heap that are obtained from running Java programs. HeapViz achieves a tree-like hierarchy in the extracted graph by using a dominator-based layout scheme. The visualization also allows collapsing and expanding nodes based on a dominator tree. The OOG supports more flexible hierarchies, due to supporting both the strict encapsulation of object domination, as well as logical containment where a child object is part of a parent object and still accessible to the outside.

Object-centric debugging. The Runtime Perspective takes an object-centric view of a system rather than a class-centric one. Similarly, object-centric debugging [16] aims for the same but for a Debugging perspective that is today predominantly stack-centric.

Whyline [11] uses a combination of static and dynamic slicing and call graphs, then enables developers to go from a list of suggested *why did* and *why didn't questions* to lines of code that did or did not get reached. Whyline requires running the system at least once, and states the questions in terms of concrete objects.

Heap abstraction. Marron et al. [13] abstract heaps, with the intent of using them for program comprehension. However, their abstraction is based on the result of a dynamic analysis. As a result, it may not reflect all the objects and relations that may be created in any possible execution.

6. Conclusion

We presented a novel Eclipse perspective, the Runtime Perspective, that complements the existing design-time perspectives in Eclipse and promotes thinking in terms of abstract objects. The Runtime Perspective moves the IDE away from focusing on a class-oriented view of the system. Instead of emphasizing tracing to class declarations, it promotes tracing from abstract objects or edges to expressions in the code.

References

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

[2] M. Abi-Antoun, S. Chandrashekar, R. Vanciu, and A. Giang. Are Object Graphs Extracted Using Abstract Interpretation Significantly Different from the Code? In *Intl. Working Conference on Source Code Analysis and Manipulation*, 2014.

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

[4] K. H. Bennett, V. Rajlich, and N. Wilde. Software evolution and the staged model of the software lifecycle. *Advances in Computers*, 56:3–55, 2002.

[5] J. Bohnet and J. Döllner. Analyzing Feature Implementation by Visual Exploration of Architecturally-Embedded Call-Graphs. In *WODA*, 2006.

[6] D. Buck, I. Diethelm, and S. Sheneman. Objektgraph: why code when MVC applications can be generated with UML-based diagrams? In *SPLASH (Companion Volume)*, 2013.

[7] H. B. Christensen. *Flexible, Reliable Software Using Patterns and Agile Development*. Chapman and Hall/CRC, 2010.

[8] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization*, 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] S. Kelley, E. Aftandilian, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. *Information Visualization*, 2012.

[11] A. J. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *ICSE*, pages 301–310, 2008.

[12] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *WCRE*, 2002.

[13] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting Runtime Heaps for Program Understanding. *TSE*, 39(6):774–786, 2013.

[14] MiniDraw. `www.baerbak.com`.

[15] ObjectAid. `www.objectaid.com`.

[16] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *ICSE*, pages 485–495, 2012.

[17] M.-A. Storey, F. Fracchia, and H. Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *J. Systems & Software*, 44(3), 1999.

[18] R. Vanciu and M. Abi-Antoun. Ownership Object Graphs with Dataflow Edges. In *WCRE*, 2012.