Finding the Missing Eclipse Perspective: the Runtime Perspective

Andrew Giang             Marwan Abi-Antoun
Department of Computer Science

Wayne State University
Outline

➢ Motivation

➢ For each feature in the Runtime Perspective, discuss:
  – closely related Eclipse views
  – new features

• Demonstration
  – Pick tasks to implement
  – Use features from the Runtime Perspective
Program comprehension is hard

- Software maintenance costs 50%--90%
  - Of that, 50% spent in program comprehension
    [Bennett et al., Advances in Computers’02]
- Different views to aid comprehension:
  - Static/code structure
  - Dynamic/runtime/execution structure
  - Other: deployment, etc.
Object-oriented design is hard

• Developers need to understand both:
  – code structure
    • it is "right there"
    • much tool support
  – runtime structure
    • different from code structure
    • some code/design patterns make differences bigger
    • less mature tool support
Structure

Code structure
E.g., class diagram

A typical object structure might look like this:

Run-time structure
E.g., object diagram
IDEs present to OO developers mainly a hierarchy of classes

• Current/popular IDEs (e.g., Eclipse) predominantly emphasize code structure:
  – Class-oriented view
  – Hierarchy of classes
• Hard for novice programmers (e.g., undergrads) to "understand" the objects
Why not present to OO developers a hierarchy of abstract objects?

- Use abstract runtime structure as a design-time perspective
  - abstract objects
  - abstract edges (relations between objects)

- Abstraction keeps things manageable
  - **Hierarchy of abstract objects**
  - Summarization of runtime objects

- Use static analysis so tool works at design time
  - Will not replace debugger (the real runtime structure)
Hierarchy of Classes vs. Objects

Hierarchy of classes
+- package/
 | +- package/
 | | +- class/
 | | | +- innerclass/
+- package/
 | +- class/
 | | +- innerclass/

Hierarchy of abstract objects
+-root/
 | +- TLD1/
 | | +- object1: B
 | | | +- MAPS
 | | | | +- hash1: Hashtable Hashtable Hashtable Hashtable
 | | | +- OWNED
 | | | | +- hash2: Hashtable Hashtable Hashtable Hashtable
 | +- TLD2/
 | | +- object2: B
 | | | +- OWNED

class B {
    public void m() {
        @Domain("OWNED")
        Hashtable hash2 = new Hashtable();
    }
}
Eclipse Runtime Perspective: missing link between Java and Debugging

Java Perspective:
• Package Explorer
• Outline View
• File/Java Search
• Type Hierarchy
• Javadoc
• Call Hierarchy
• Class Diagrams

Debugging Perspective:
• Call stack
• Watch window

Runtime Perspective:
• Abstract Object Tree
• Classes behind interface
• Summary View
• Abstract Stack
• Partial Graph View
• Related Objects and Edges
Debugger

• Shows runtime objects (specific instances) of a program in execution.
• Limitations: too many specific instances that may not even matter for most tasks
At runtime, object-oriented program appears as a Runtime Object Graph

runtime object

runtime edge
Abstract zero or more runtime objects into an abstract object
Abstract object represents a role

• **Abstract object** merges objects that have same role, i.e., `<type, domain, parent type>`
  – Domain: named, conceptual **group** of objects
  – Each object can have nested domains
  – Achieve hierarchy of abstract objects
• With hierarchy, collapse objects under other objects
  – Architecturally significant objects near top of hierarchy
  – Implementation details (data structures) further down
  – High-level understanding and detail
Abstract edges show relations between abstract objects
Different types of abstract edges are extracted

• Points-to edges (PT) due to field references
• Dataflow edges (DF) due to usage
  – field read
  – field write
  – method invocation
• Some edges can refer to objects
  \[ o_1 : C_1 \rightarrow o_2 : C_2 \quad [o : O] \]
Extracting the Runtime Perspective

• Add annotations and type check them
• Extract abstract graph with static analysis
  – Save abstract graph to external file
  – File maintains traceability to code
• Switch to Runtime Perspective
  – Loads the abstract graph from external file
  – Mines the abstract graph
  – Displays information in various views
Using the Runtime Perspective

1. Abstract Object Tree
2. Abstract Stack
3. Summary View
4. Partial Graph
5. Related Objects & Edges
For each Eclipse view

• Discuss limitations
• Discuss closely related view from the Runtime Perspective:
  – main features
  – how it complements existing information sources
Code vs. Runtime Perspective

Java Perspective:
• Package Explorer
• Outline View
• File/Java Search
• Type Hierarchy
• Javadoc
• Call Hierarchy
• Class Diagrams

Debugging Perspective:
• Call stack
• Watch window

Runtime Perspective:
• Abstract Object Tree
• Classes behind interface
• Summary View
• Abstract Stack
• Partial Graph View
• Related Objects and Edges
Eclipse Package Explorer

- Hierarchy of classes
- Organized by package
  - Note: a class cannot contain a package

- Limitation: packages, classes/interfaces sorted alphabetically.
Abstract Object Tree

- Hierarchy of abstract objects and domains
Abstract Object Tree Usefulness

• Search for abstract objects by name, type:
• Trace to code:
  – Trace to expressions – not declarations
  – From object to object creation expression
  – From edge to field read, field write, etc.
Demo

- Switch to Runtime Perspective
- Go to Abstract Object Tree
  - Talk about Main object
  - Three top-level domains
  - Low-level objects further down
- Search for: game
- Trace to code
Call Hierarchy

- Shows caller and callees starting from a selected method
- Limitation: traces to method invocations
Abstract Stack

• Each abstract object is due to an object creation expression in the code
• From abstract object, select the abstract stack
Abstract Stack Usefulness

• Usefulness: show the nested abstract interpretation contexts that lead to the creation of an abstract object.
  – Expose notion of "object sensitivity" in program analysis to developers
  – Make explicit the receivers in "call stack"
Abstract Stack Example

A a = new A();
a.mA();

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

• Switch to Runtime Perspective
• Go to Abstract Object Tree
• Search for: boardDrawing
• Go to Abstract Stack trace
Javadoc

• Documentation from source code comments generated to HTML.
• Limitations: Shows packages, classes, and methods listed alphabetically; no ranking by relevance.
Summary View

• Shows a ranked list of:
  • **MIC** - Most Important Classes
  • Given C a fully qualified name of a class
    – **MIRC**(C): Most Important Related Classes to C
    – **MIM**(C) - Most Important Methods of C
    – **MCBI**(C) - Most Important Classes Behind an Interface
Summary View Usefulness

• Summary View gives developer a ranked list of classes and methods.
  – Rankings gathered from multiple strategies that traverse the abstract graph
  – Unlike Package Explorer or Javadoc, where packages, classes and methods organized alphabetically
Demo

• Switch to Runtime Perspective
• Select class: GameStub
• Look at MIRCs, MIMs
• Select a class from the list
• See the Summary View update.
Related Objects and Edges

- Find related places in the code related to this line of code
- See all objects and edges related to the currently selected ASTNode
Related Objects and Edges

Usefulness

• Usefulness: identify all the code elements that map to the same abstract object or the same abstract edge
  – Similar to notion of "impact analysis"

• Unlike Summary View, does not lift information back to types
  – Traverse graph of abstract objects and edges
Demo

• Switch to Runtime Perspective
• Select class: BoardDrawing
• Select figureMap.put() method invocation in the editor
• Look at the related edges
• Trace to code to another related edge
Graph View

- Displays a partial runtime graph
- Using MIRCs: show related nodes and edges from the class that the developer is working on
Graph View Usefulness

• Graphical representation
  – Can get cluttered
  – Most useful for top-level objects
• Can expand/collapse objects on demand
• Can hide/show objects on demand
  – Objects NOT deleted; just hidden from graph
Demo

• Switch to Runtime Perspective
• Show all objects in the shared domain
  – Root object
• Hide object in shared
• Expand main object
Outline

• Motivation

• For each feature in the Runtime Perspective, discuss:
  – closely related Eclipse views
  – new features

➢ Demonstration
  ➢ Pick tasks to implement
  ➢ Use features from the Runtime Perspective
Subject System

- MiniDraw framework (1400 LOC)
  - Pedagogical object-oriented framework
  - Support development of board games
  - Uses many design patterns
- Breakthrough: game created using framework
  - Game similar to chess or checkers
  - Objective of game: reach the opponents home row
Task #1

- Validate piece movement
- Board Piece can move one square straight or diagonally towards the opponents home row.
Task #2

- Implement the capture of a board piece
- A board piece can only capture another board piece on a diagonal move.
- Piece takes position of the captured piece
Task #3

- Implement an undo feature
- Menu item “Undo move”
Task #4

- Implement a status bar
- Status bar is built into the framework
- Update status bar on piece movements
How ranking classes works
Computing MICs, MIMs, MIRCs

• Most Important Classes: MIC
• Use the hierarchy of objects
  – All top-level abstract objects are included
  – One strategy: rank classes of those objects based on number of incoming/outgoing edges to abstract object of that type
• Most Important Methods: MIM(C)
• For all method declarations of a given class
  – Rank methods by the number of edges due to method invocations in the abstract graph
Computing MICs, MIMs, MIRCs

• Most Important Related Classes: MIRC(C)

• From a given Class C
  – Traverses the set of edges filtered by edge type (dataflow, points-to, etc)
  – Collect the union of edges having a source or destination that is an object of type C or a subtype of C

• Other strategies are possible
Computing MCBIs

• Most Important Classes Behind an Interface: MCBI(C)
  – Find reachable domains from that location (i.e., actual domains bound to formal domain parameters)
  – Find all classes in those domains
  – For all classes that implement the interface, rank these classes using strategy similar to MIRCs
Related Tools

• Code exploration tools:
  – Focus on code structure, not runtime structure
  – Jadeite [Stylos et al., VL/HCC'09]: ranks element in JavaDoc by relevance from a web query (search popularity)

• Heap exploration tools:
  – Backwards/reversible debugging
  – Mine concrete heaps
  – Notion of time (missing here)

• Abstract heap tools [Marron et al., TSE'13]
  – Tradeoffs of static vs. dynamic analysis
Future Work

• Evaluate the tool in user studies
  – Replicate results from previous experiment
    [Ammar and Abi-Antoun, WCRE’12]

• Use the tool in educational setting
  – Beginners learning design patterns, etc.
  – Use in laboratory component of course

• Use metrics to identify when runtime structure really different from code structure
Conclusion

• Runtime Perspective complementary to:
  – Java Code Perspective and
  – Debugging Perspective

• Mine information from an abstract runtime structure into several views
  – Abstract object tree
  – Abstract stack
  – Summary view
  – Related objects and edges