

Finding the Missing Eclipse Perspective: the Runtime Perspective

Andrew Giang

Marwan Abi-Antoun

Department of Computer Science



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.

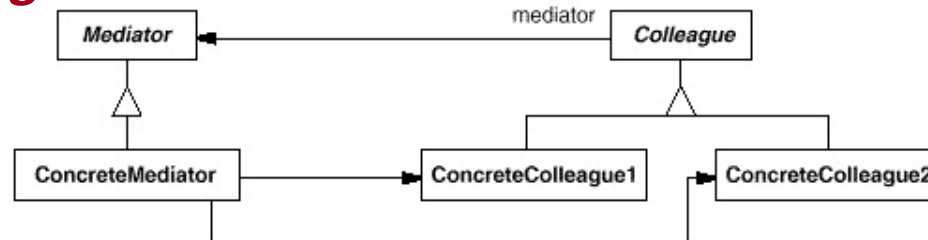
Object-oriented design is hard

- 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

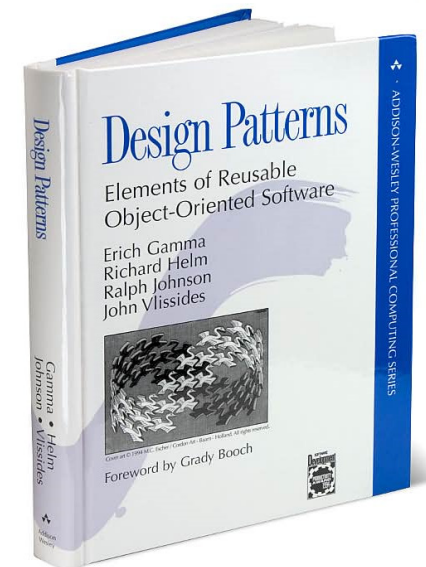
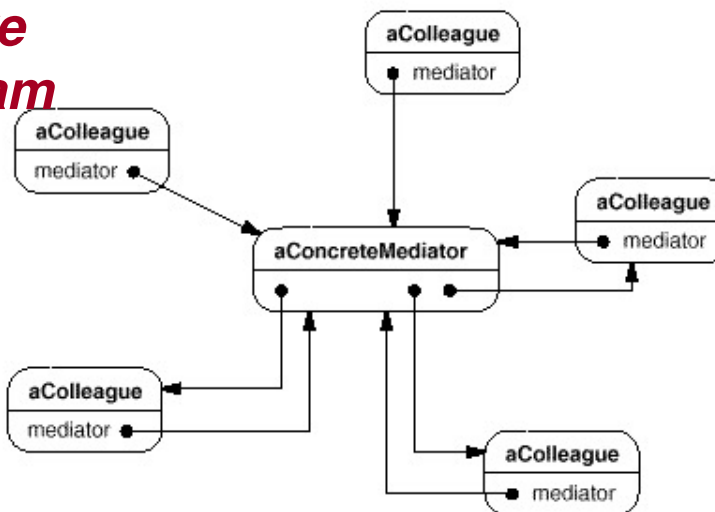
E.g., class diagram



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/
```

Trace to code

```
package x;  
class Hashtable {  
  ...  
}
```

Hierarchy of abstract objects

```
+--root/  
  +- TLD1/  
    | +- object1: B  
    |   +- MAPS  
    |   +-   
    |   +- OWNED  
    |   +-   
  +- 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:

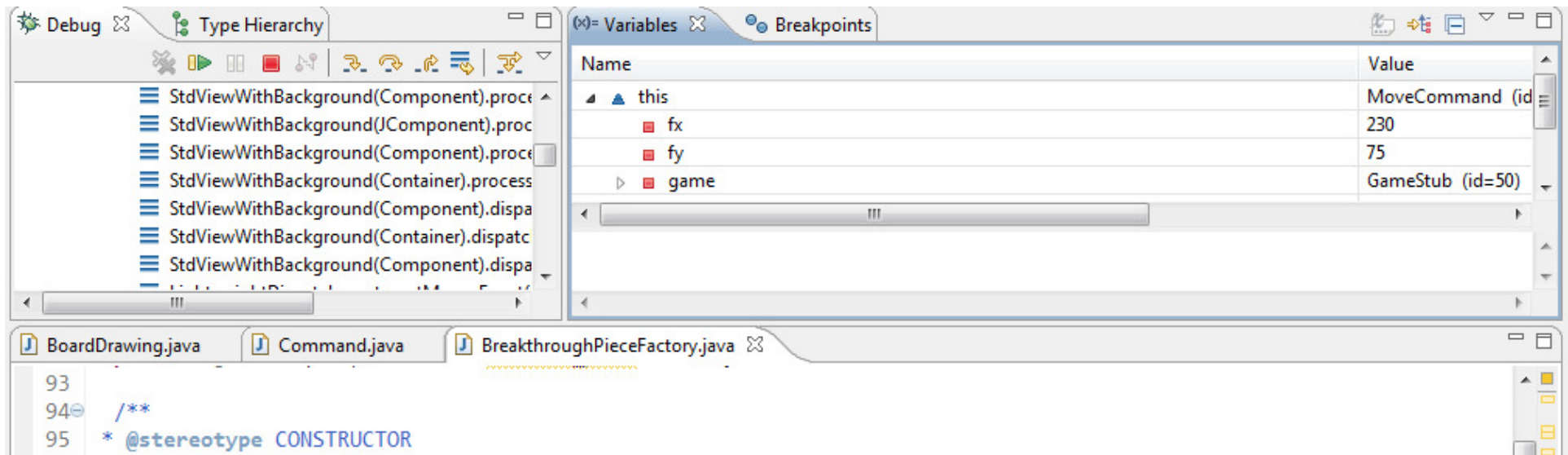
- Call stack
- Watch window

Runtime Perspective:

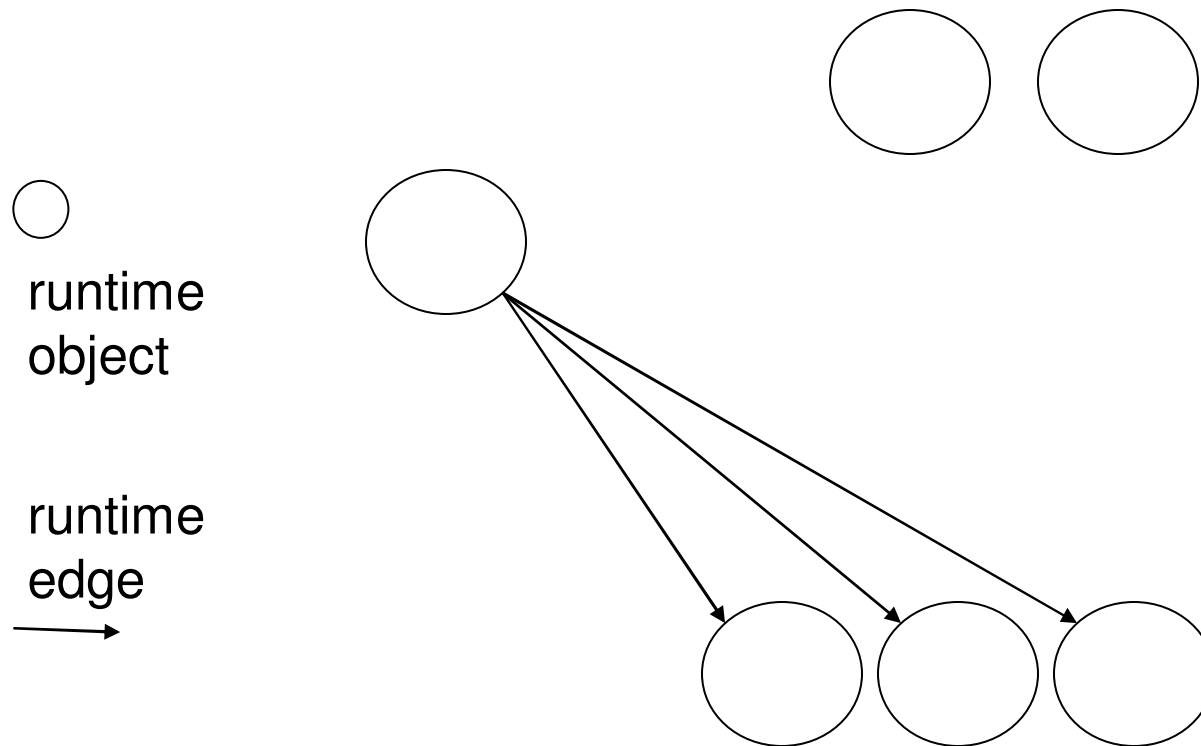
- Abstract Object Tree
- Classes behind interface
- Summary View
- Abstract Stack
- Partial Graph View
- Related Objects and Edges

Debugger

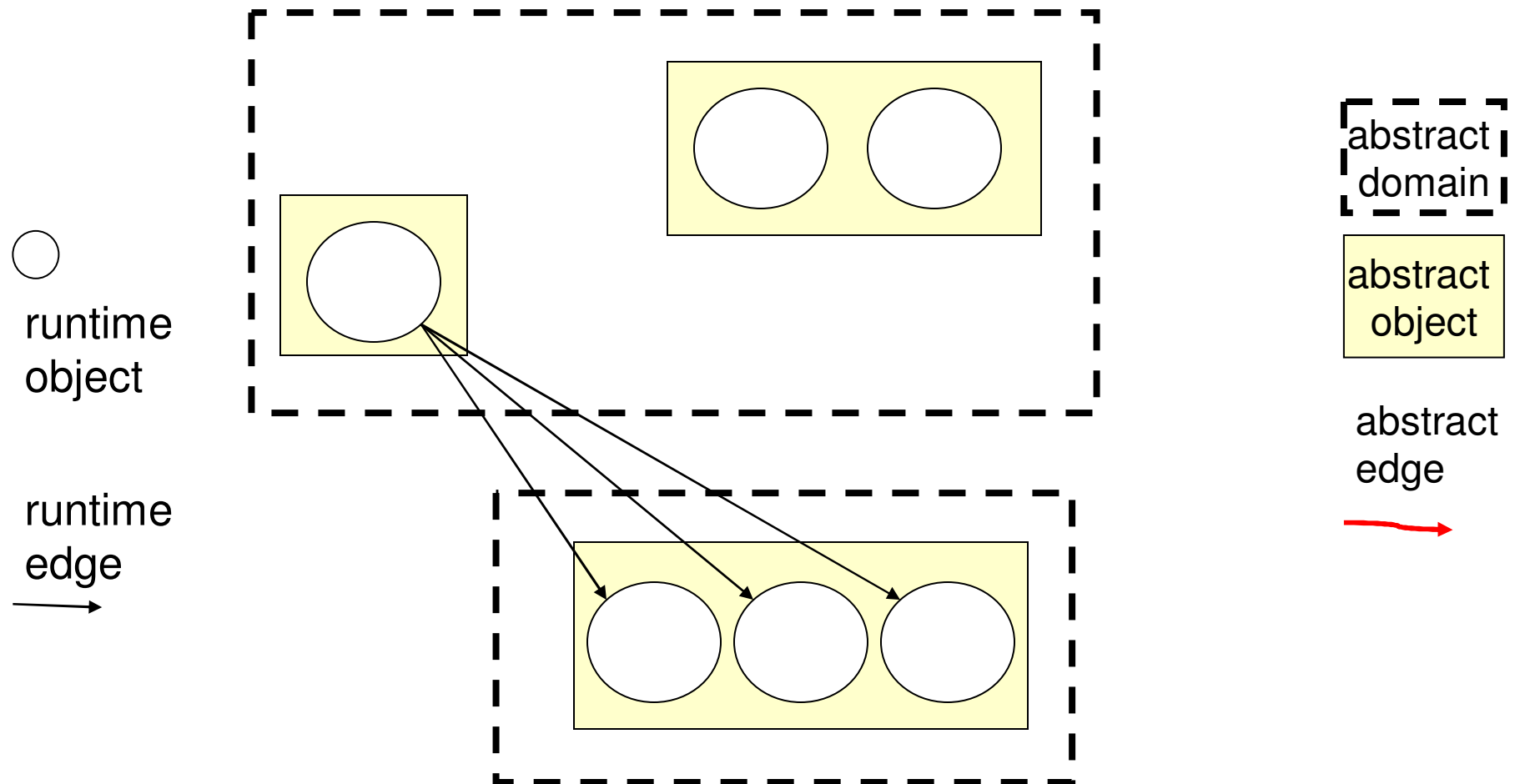
- Shows runtime objects (specific instances) of a program in execution.
- Limitations: too many specific instances that may not even matter for most tasks



At runtime, object-oriented program appears as a Runtime Object Graph



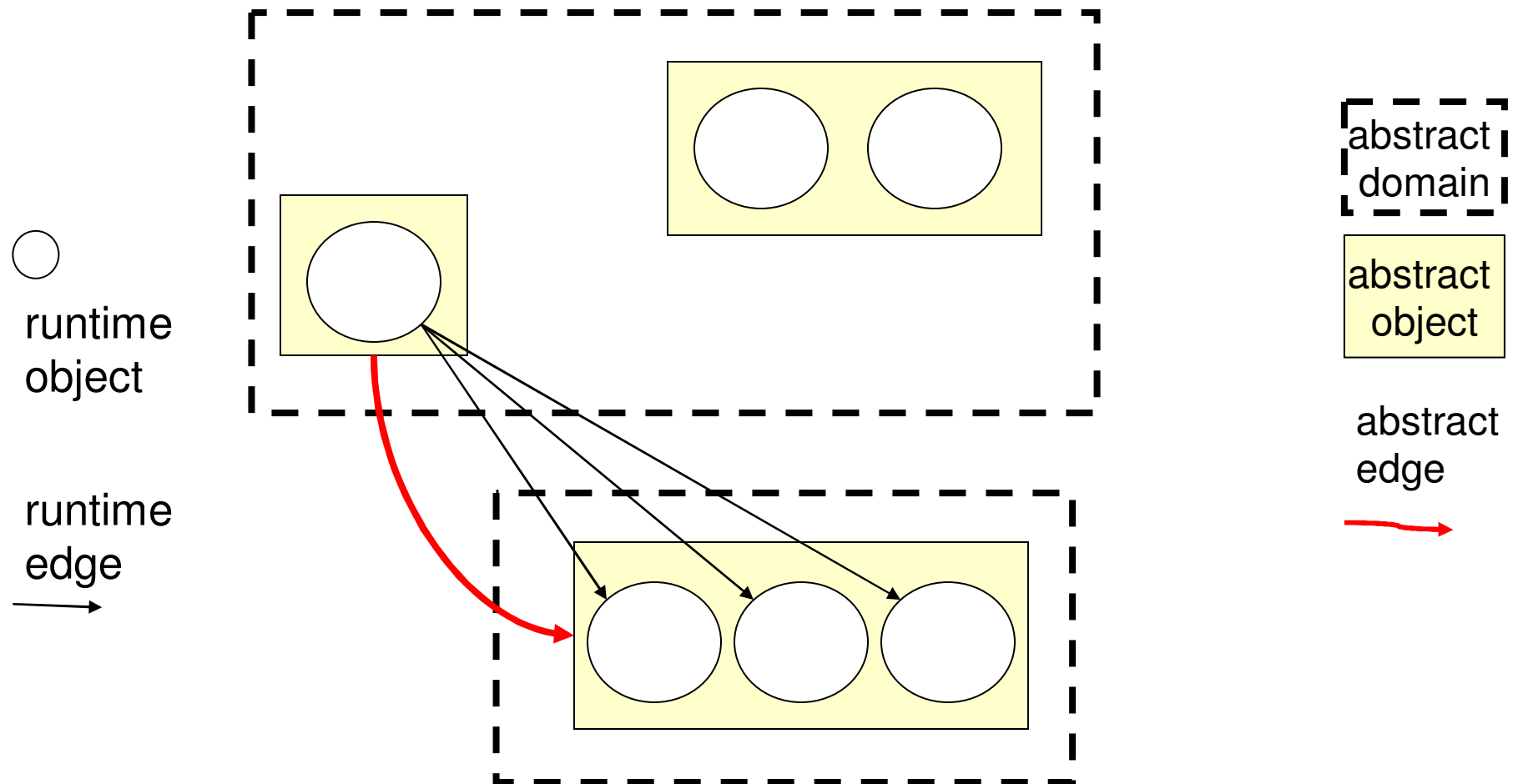
Abstract zero or more runtime objects into an abstract object



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
$$o1 : C1 \rightarrow o2 : C2 \quad [o : O]$$

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

The screenshot displays the Eclipse IDE in the Runtime Perspective. The central editor shows the `BreakThroughMain.java` file with the following code:

```

42
43 /**
44  * @stereotype FACTORY
45  * @stereotype COLLABORATOR
46  */
47 @Domain("unique<D,unique<D>>") Map<Position, List<BoardFigure>> generatePieceMultiMap
48
49 @Domain("unique<D,unique<D>>")
50 Map<Position, List<BoardFigure>> m = new HashMap<Position, List<BoardFigure>>();
51 for ( int row = 0; row < 8; row ++ ) {
52     for ( int col = 0; col < 8; col ++ ) {
53
54         @Domain("D") Position p = new Position(row,col);
55         int whatIsOnThisSquare = game.get(p);
56         @Domain("unique<D,U,L,D>>")
57         List<BoardFigure> l = new ArrayList<BoardFigure>();
58         if ( whatIsOnThisSquare != Game.NONE ) {
59             @Domain("lent")
60             String gifname = whatIsOnThisSquare == Game.WHITE ?
61                 Constants.WHITE_PAWN_IMAGE_NAME :
62                 Constants.BLACK_PAWN_IMAGE_NAME;
63
64             @Domain("l<U,L,D>") MoveCommand command = new MoveCommand(game).

```

The interface includes several panels:

- Package Expl**: Shows the package structure with a filter text field.
- Abstract Object Tree**: Displays a tree of objects, including `BoardFigure`, `BoardDrawing`, and `Position`.
- Abstract Stack**: Shows a stack of objects, including `BoardFigure`, `BoardDrawing`, and `Position`.
- Summary View**: Displays a table of objects and their weights.
- Partial OOG**: Shows a partial graph of objects and edges.
- Related Objects and Edges**: Shows a list of related objects and edges.

(1) Abstract Object Tree

(2) Abstract Stack

(3) Summary View

(4) Partial Graph

(5) Related Objects & Edges¹⁷

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:

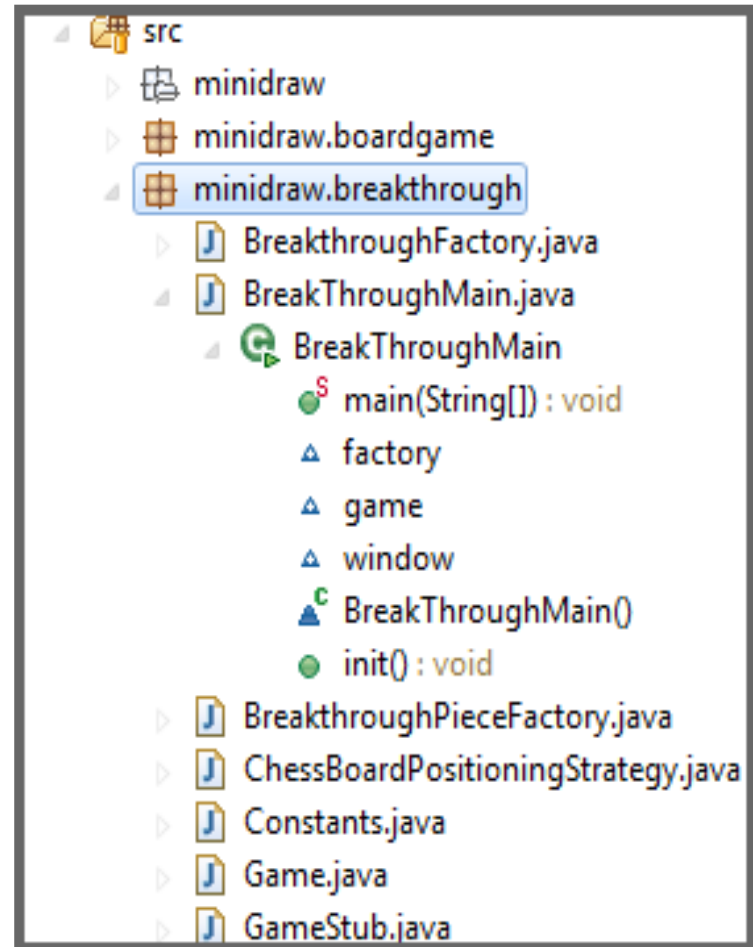
- Call stack
- Watch window

Runtime Perspective:

- Abstract Object Tree
- Classes behind interface
- Summary View
- Abstract Stack
- Partial Graph View
- Related Objects and Edges

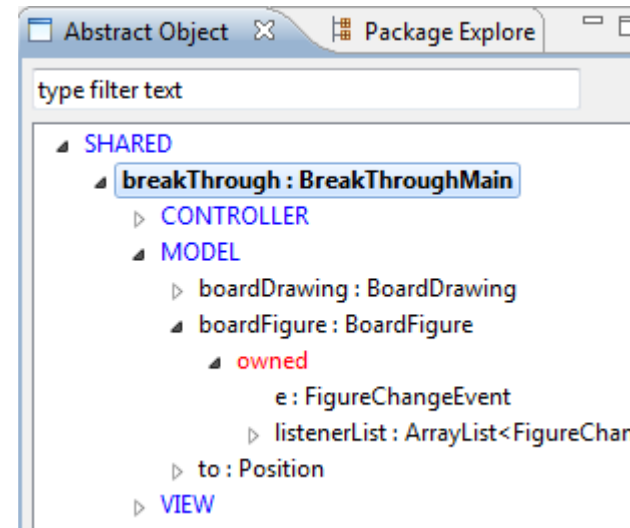
Eclipse Package Explorer

- 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



- Domain
- Object
- BOLD Main Object
- owned Domain

Abstract Object Tree Usefulness

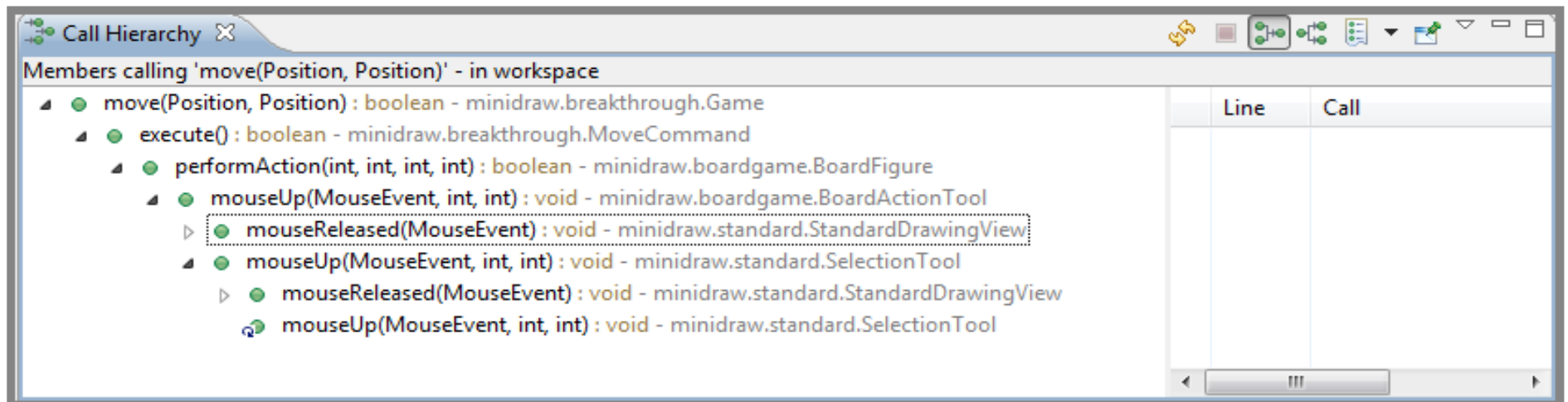
- Search for abstract objects by name, type:
- Trace to code:
 - Trace to expressions – not declarations
 - From object to object creation expression
 - From edge to field read, field write, etc.

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

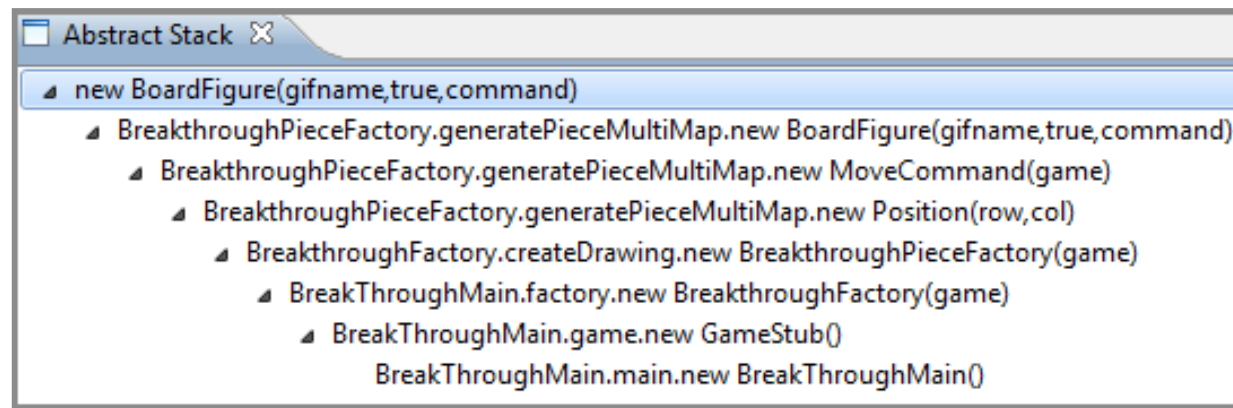
Call Hierarchy

- 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



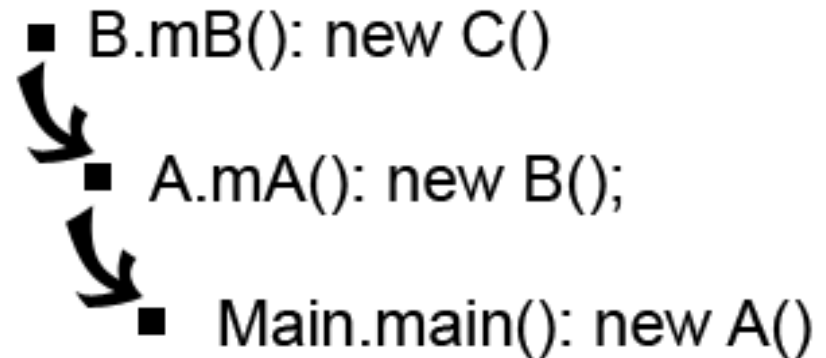
Abstract Stack Usefulness

- 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

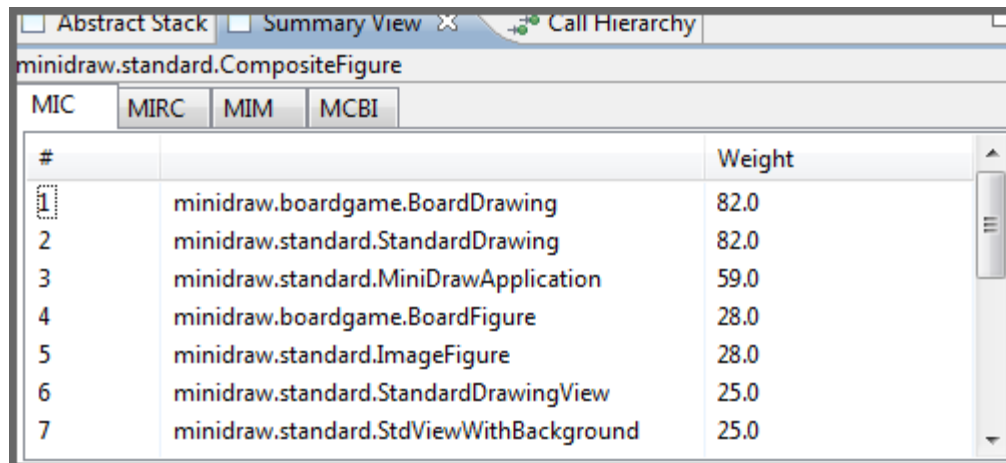
- Documentation from source code comments generated to HTML.
- Limitations: Shows packages, classes, and methods listed alphabetically; no ranking by relevance

The screenshot displays a Javadoc web page for the `minidraw.framework` package. The left sidebar contains a navigation menu with links to 'All Classes' and 'Packages'. The 'All Classes' list includes `AbstractFigure`, `AbstractTool`, `BoardActionTool`, `BoardDrawing`, `BoardFigure`, `BoardGameObserver`, `Command`, `CompositeFigure`, `DragTracker`, `Drawing`, `DrawingChangeEvent`, `DrawingChangeListener`, `DrawingChangeListenerHandler`, `DrawingEditor`, `DrawingView`, `Factory`, `Figure`, `FigureChangeEvent`, `FigureChangeListener`, and `FigureFactory`. The main content area shows the `minidraw.framework` package and the `Interface Figure`. It lists 'All Known Implementing Classes' as `AbstractFigure`, `BoardDrawing`, `BoardFigure`, `CompositeFigure`, `GroupFigure`, `ImageFigure`, and `StandardDrawing`. Below this, the `public interface Figure` is declared. A descriptive paragraph explains the interface's role and responsibilities. At the bottom, a 'Method Summary' table is partially visible.

Method Summary	
void	<code>addFigureChangeListener (FigureChangeListener l)</code> Adds a listener for this figure.
void	<code>changed ()</code>

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



#		Weight
1	minidraw.boardgame.BoardDrawing	82.0
2	minidraw.standard.StandardDrawing	82.0
3	minidraw.standard.MiniDrawApplication	59.0
4	minidraw.boardgame.BoardFigure	28.0
5	minidraw.standard.ImageFigure	28.0
6	minidraw.standard.StandardDrawingView	25.0
7	minidraw.standard.StdViewWithBackground	25.0

Summary View Usefulness

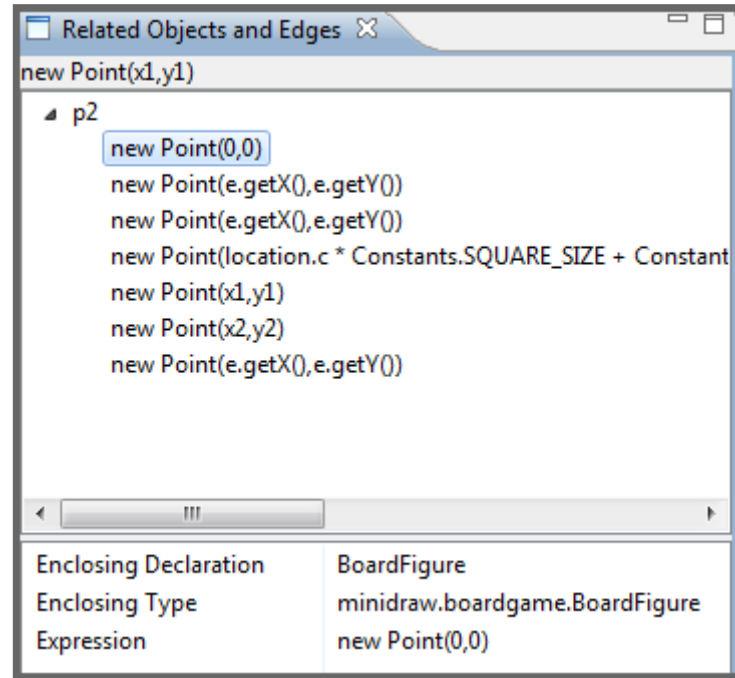
- Summary View gives developer a ranked list of classes and methods.
 - Rankings gathered from multiple strategies that traverse the abstract graph
 - Unlike Package Explorer or Javadoc, where packages, classes and methods organized alphabetically

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

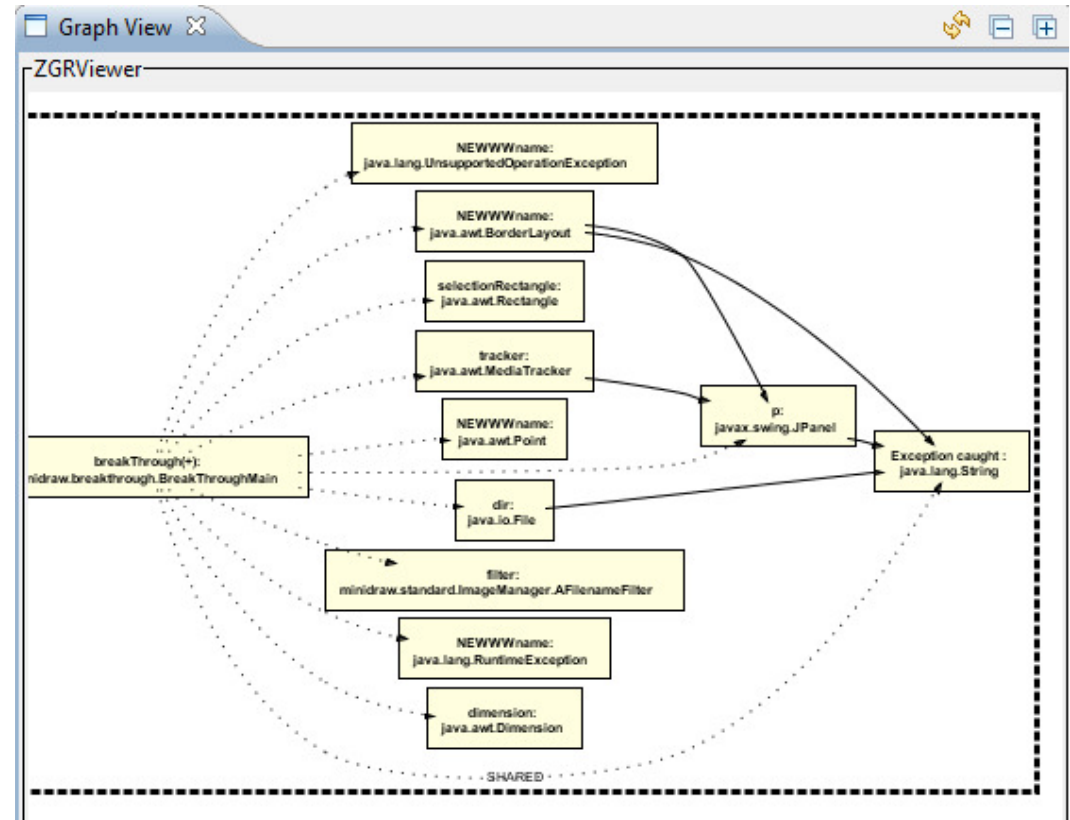
- Usefulness: identify all the code elements that map to the same abstract object or the same abstract edge
 - Similar to notion of "impact analysis"
- Unlike Summary View, does not lift information back to types
 - Traverse graph of abstract objects and edges

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

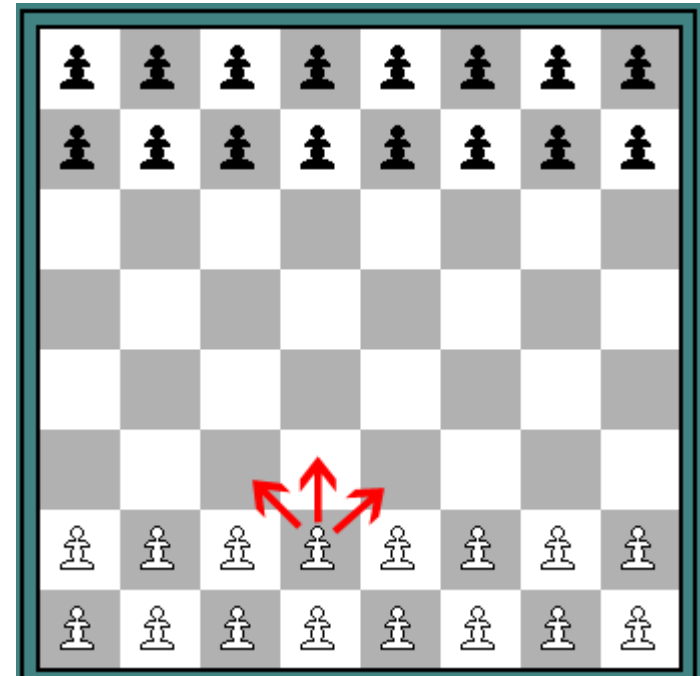
- Motivation
- For each feature in the Runtime Perspective, discuss:
 - closely related Eclipse views
 - new features
- Demonstration
 - Pick tasks to implement
 - Use features from the Runtime Perspective

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

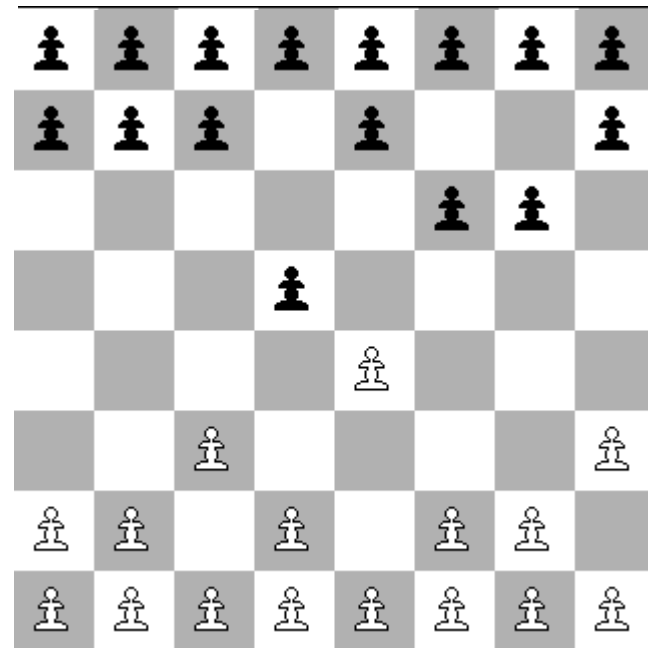
Task #1

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



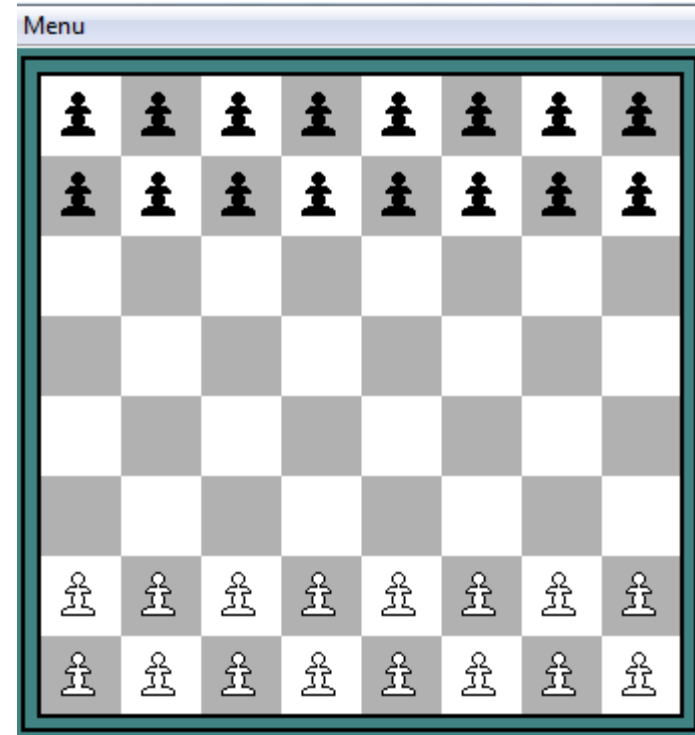
Task #2

- 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



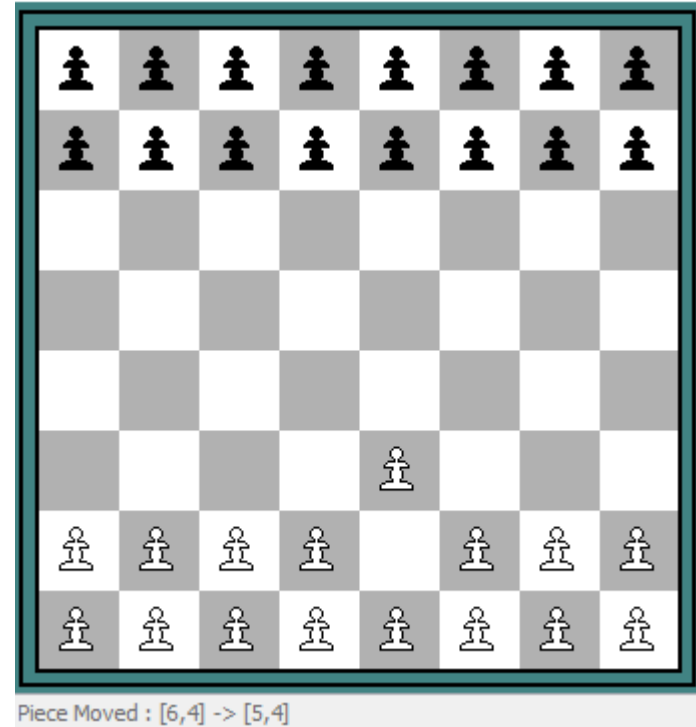
Task #3

- 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