

The Eclipse Debug Framework

Darin Wright and Michael Rennie
IBM Rational Software

March 20th, 2006

What You Need

- To get the most from this tutorial, you need to bring a laptop capable of running Eclipse and developing plug-ins.
- Before the tutorial, you need to have the following software loaded on your machine:
 - Eclipse 3.2 M5 <http://www.eclipse.org/>
 - Perl (ActivePerl) <http://www.activestate.com/Products/ActivePerl/>
 - Tutorial source <http://www.eclipsecon.org/>
- Eclipse should be installed and running.
- Perl should be installed and running.
- Source code zip should be unzipped and ready to use.

Tutorial Example

- One example to work towards
 - Contrast each module with example as tutorial proceeds
- Architecture of Example
 - Assembly language with an interpreter written in Perl
 - Send request to the interpreter over one socket
 - Read events over another socket
 - Abstraction of **IProcess** is the Perl interpreter
 - Abstraction of **IDebugTarget** is the program being interpreted

Plugin Architecture

- Eclipse is arranged as set of plugins
 - Each plugin provides separate functionality
 - Plugins are intended to define a specific domain, with no over-lapping or interdependence
- A good standard convention for plugin design includes
 - A core plugin – the guts of your plugins, what it does
 - A UI plugin – the user interface for your plugin
 - E.g. `org.eclipse.debug.core` and `org.eclipse.debug.ui`, both together make the ‘debug plugin’

Module Structure

- Tutorial arranged as separate modules
 - Each modules follows a standard format
- Each module contains
 - Introduction – what we want to accomplish in each module
 - The players – key infrastructure components and their interactions
 - Check list – steps to follow when building your debugger
 - Detailed descriptions – further insight into specific aspects of components of the module
 - Exercise – Exercises to demonstrate the major points of the module

Tutorial Modules

- The Basics
 1. Launching
 2. Debug model
 3. Breakpoints
 4. Source lookup

- The Advanced
 5. Variable customizations
 6. Custom views and actions
 7. Editor-debugger interactions

Module 1

The Launch Framework

Introduction

- The launch framework includes facilities for:
 - Spawning an O/S process
 - Persisting information about how something is launched
 - A framework for editing launch parameters (GUI)
 - An extensible set of launch modes (run, debug, profile...)
 - Selection sensitive actions for launching
- To run or debug code under development is fundamental to an IDE
 - Eclipse's launching capabilities depend entirely on the current set of installed plug-ins
- For the purpose of this tutorial launching initiates a debug session

The Launch Players

- Infrastructure
 1. Launch configuration types – available launch types
 2. Launch configurations – description of how and what to launch
 3. Launch manager – stores types, configurations and launch objects
 4. Launch delegates – performs the launch
 5. Launch objects – container of launched processes/debug targets

- UI
 6. Tab groups – user interface for editing launch configurations
 7. Launch shortcuts – contextual action for launching

1. Launch Configuration Types

- Each launch configuration is of a specific type
 - e.g. Java™ Application
- All launch types are available from the launch manager
- The `<launchConfigurationTypes>` extension point allows new launch types to be contributed to the platform
- The platform provides the only implementation of launch configuration types – `ILaunchConfigurationType`.
 - Each extension contributes a launch delegate (more later)
- Each launch type has domain specific attributes associated with it, which are stored in individual launch configurations
 - E.g. Java applications have attributes like main type whereas PDE has a set of plug-ins

2. Launch Configurations

- Description of how and what to launch
 - Persisted map of keys and values
 - The platform provides the only implementation of launch configurations – **ILaunchConfiguration** and **ILaunchConfigurationWorkingCopy**
 - A launch configuration is read-only – a working copy is used to edit a configuration in a transaction-like manor
 - A working copy is created from a launch configuration
 - Launch tabs modify a working copy and can commit changes to the original or revert
- Designed to be shared across different launch modes
 - For example, a launch configuration describes how to launch a Java application (main type, etc.), but can be launched in run, debug, or other modes

2. Launch Configurations Continued

- The Debug platform defines/contributes three launch modes
 - Run, Debug, and Profile
 - The set is extensible via the `<launchModes>` extension point
- The launch configuration (state) and launch delegate (behavior) are separated to promote lazy loading of plug-ins.
 - The debug platform provides one implementation of launch configurations so they can be loaded and visible in the launch history without loading the launch delegates until/if they are needed to perform an actual launch.

2. Launch Configurations Continued

- **New to 3.2:** launch configurations can be mapped to a set of resources
 - Facilitates resource based filtering in the launch configuration dialog, launch history and favorites menu
 - Clients must manage the resource mapping
- **New to 3.2:** launch configurations can be migrated
 - Facilitates upgrading existing launch configurations to support new features
 - Contributed by new optional `<migrationDelegate>` attribute on the launch configuration types extension point
 - Must implement `ILaunchConfigurationMigrationDelegate`

3. Launch Manager

- Manages all launch configurations, configuration types and launches
 - Can be queried for all available launch configurations
 - Can be queried for all registered configuration types
 - Can be queried for all registered launch modes
 - Launches are registered/deregistered with the manager
 - Provides change notification for launches
 - Also manages all registered source container types, source path computers and creates source locaters (module 4)

4. Launch Delegates

- A launch configuration type contributes a launch delegate for specific launch modes
 - Implements **ILaunchConfigurationDelegate**
 - E.g. the Java debugger contributes a launch delegate for run and debug modes for Java applications
- The `<launchDelegates>` extension point allows a launch delegate to be contributed to an existing launch configuration type for a specific launch mode
 - E.g., a tool can contribute a launch delegate for launching Java applications in profile mode, even though the base SDK does not support profiling.
 - Allows launch configurations to be extended by other tools
 - `<launchConfigurationTabGroups>` extension point provides a `<mode>` attribute to allow a tab group to be contributed for a specific mode

2. Launch Delegates Continued

- The debug platform provides an abstract launch delegate that should be sub-classed.
 - The abstract delegate (**LaunchConfigurationDelegate**) provides infrastructure to perform:
 - Scoped builds before launching, allowing subclasses to specify the set of projects that should be compiled.
 - Scoped search for errors/problems, allowing the launch to be aborted
 - **New to 3.2** - Scoped search for unsaved editors, allowing the editors to be saved

5. Launch Objects

- Container of processes and debug targets created by launching
 - The debug platform provides a standard implementation of **IProcess** based on `java.lang.Process`
 - Provides convenience methods for launching/creating an **IProcess** from a command line
 - Allows you to provide your own implementation if desired
- The debugger provides a console to display the standard I/O streams of a process
 - For each **IProcess** added to an **ILaunch**, the debug platform allocates a console attached to its I/O streams
 - **std.out** and **std.err** are written in blue and red
 - Keyboard is attached to **std.in** - input is buffered and written to **std.in** when <Enter> is pressed

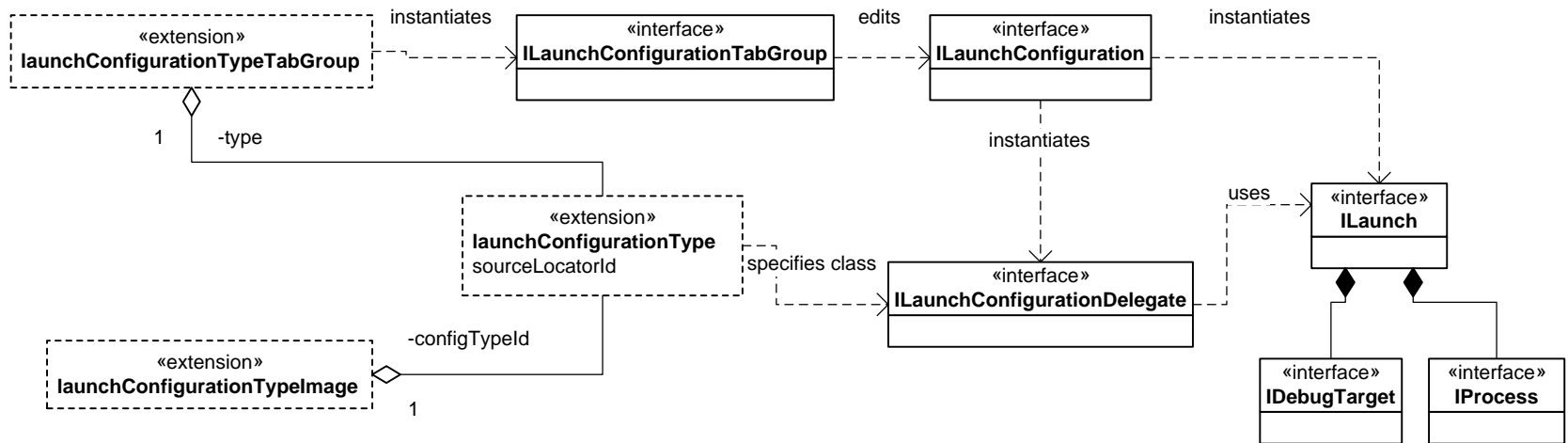
6. Tab Groups

- A set of tabs used to display and edit a single launch configuration
 - Displayed in the launch dialog when a configuration is selected
 - Tabs operate on a launch configuration working copy in a transaction-like manor – can commit or revert
- Contributed via the `<launchConfigurationTabGroups>` extension point
 - Often the same set of tabs is used for all launch modes, but the extension point provides a `<mode>` attribute to allow a tab group to be contributed for a specific mode

7. Launch Shortcuts

- Launch shortcuts are added to the Run As... context menu
 - Provides a simple way for users to launch a file/program
 - Creates a configuration (if one does not already exist), and launches it
 - Contributed via the `<launchShortcuts>` extension point
- Launch shortcuts provide enablement expressions
 - Specify when it should be enabled for a selection in the workbench (i.e. appear in the Run As... context menu)
 - Enablement expressions are XML boolean expressions that are common to many extension points
 - The debug platform provides some property testers
 - Filename pattern matching, file content type, project nature, etc.

The Launcher Model



Checklist: Create Your Launcher

1. Create an implementation of `ILaunchConfigurationDelegate`
2. Create an implementation of `ILaunchConfigurationTabGroup`
3. Define the `<launchConfigurationType>` extension to point to (1)
4. Define the `<launchConfigurationTypeImage>` extension to point to (3)
5. Define the `<launchConfigurationTabGroup>` extension to point to (2) and (3)
6. Define a `<launchShortcut>` extension and its implementation of `ILaunchShortcut`

Exercise 1: Extensions & Launch Object

- Goal:
 - Experience the interaction of launch configuration types, tabs, and delegates by defining the extension points that connect the players and coding a delegate to launch an O/S process.

- We provide:
 - A set of plug-ins with stubs for the different players

- Your mission:
 - Define extensions
 - Launch configuration type
 - Launch configuration type image
 - Tab group

Exercise 1: Continued

- Your mission (cont'd)
 - Code: tab interaction
 - Add tabs to the tab group
 - Update tab with attributes from a launch configuration (read)
 - Update a launch configuration with data from the tab (write)
 - Validate data in the tab
 - Code: launch
 - Write a simple delegate that launches an O/S process to write text to a command shell

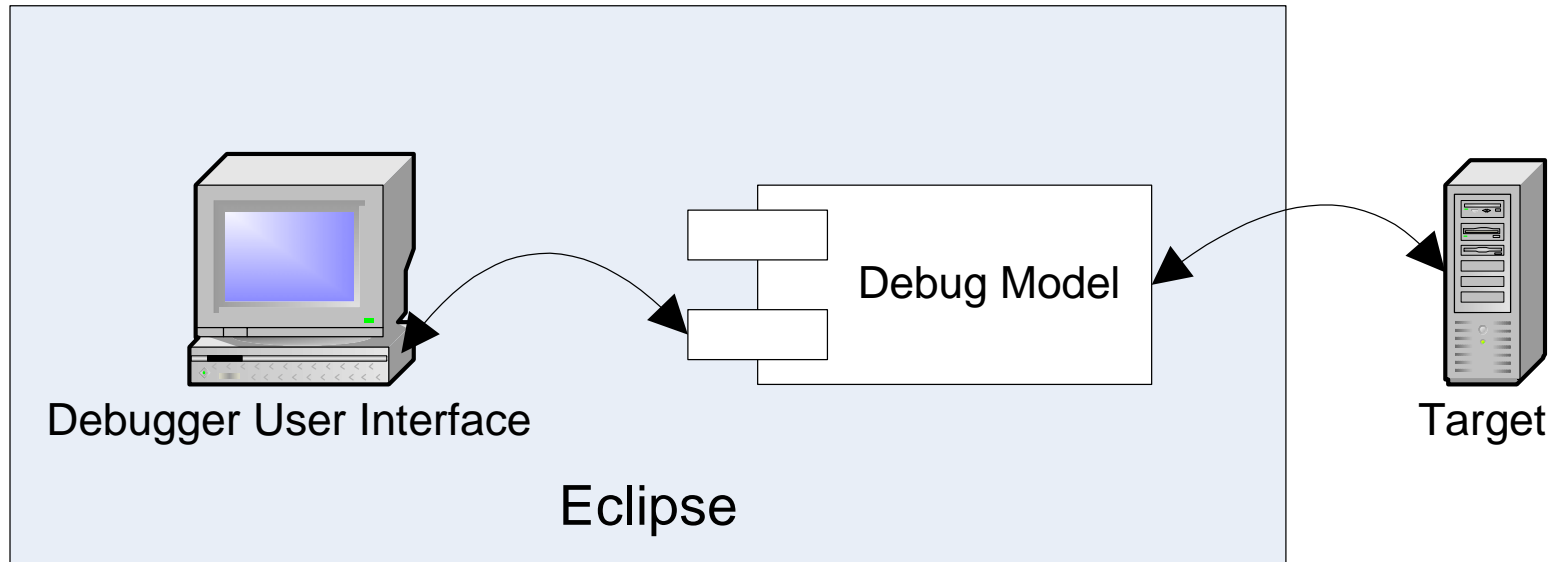
Module 2

The Debug Model

Introduction

- Why have a Debug Model?
- Well, the debugger has to have some representation of the system.
 - I.e., to debug with abstractions rather than bits and bytes, we have to define those abstractions.
- The Eclipse Debug Model contains the basic abstractions of most imperative execution environments.
 - Process, Thread, Stack Frame, Variable, Breakpoint, ...
- The Eclipse Debug UI interacts with these abstractions and the different plug-in debuggers provide the implementations.

Introduction Continued



The Debug Model Players

- Infrastructure
 1. Debug Model Elements – the program being debugged
 2. Capabilities – support for stepping, resuming, terminating, etc.
 3. Debug Events – describe happenings in an executing target or process

- UI
 4. Views – Display the debug session – threads, frames, variables, etc.
 5. Actions – interact with the program being debugged – step, resume, etc.
 6. Debug Model Presentation – provides labels and images for debug model elements

1. The Debug Model Elements

1. The standard debug model contains

- Debug Target
 - Threads
 - Stack Frames
 - Variables
 - Register Groups

IDebugTarget

IThread

IStackFrame

IVariable

IRegisterGroup

2. Variables, in this context contain

- Values (**IValue**), which can contain other variables to represent complex data structures

3. A register group contains

- Registers (**IRegister**), which are just variables

2. Capabilities

- The standard debug capabilities are
 - Step (over, into, return) **IStep**
 - Terminate **ITerminate**
 - Suspend & Resume **ISuspendResume**
 - Disconnect **IDisconnect**
 - Drop to Frame **IDropToFrame**
- The standard debug elements implement standard capabilities
 - `IDebugTarget` **extends** `ITerminate`, `ISuspendResume`, `IDisconnect`
 - `IThread` **extends** `ITerminate`, `ISuspendResume`, `IStep`
 - `IStackFrame` **extends** `ITerminate`, `ISuspendResume`, `IStep`
- The debug toolbar buttons/actions operate against these interfaces

3. Debug Events

- A debug event describes something that has happened in a program being debugged or in a running process.
 - An event has a type (kind) and detail code
- The user interface requires debug model elements and process implementations to generate the debug events. For example:
 - **IProcess** – CREATE, TERMINATE
 - **IDebugTarget** – CREATE, TERMINATE, SUSPEND, RESUME
 - **IThread** – CREATE, TERMINATE, SUSPEND, RESUME
- Required events are specified in the **DebugEvent** class

3. Debug Events Continued

- Detail codes describe why an event occurred:
 - A suspend could be caused by: STEP_END, BREAKPOINT, CLIENT_REQUEST, EVALUATION, EVALUATION_IMPLICIT
 - A resume could be caused by: STEP_INTRO, STEP_OVER, STEP_RETURN, CLIENT_REQUEST, EVALUATION, EVALUATION_IMPLICIT
- The debug platform provides event notification
 - `DebugPlugin.fireDebugEventSet(DebugEvent[] events)`
 - `DebugPlugin.addDebugEventListener(IDebugEventSetListener listener)`

4. Debug Views

- The Debug views display debug model elements
 - Update in response to debug events
- The standard views
 - Debug, Variables, Registers, Expressions, Breakpoints, Console
- Note: when a view refreshes it attempts to maintain selection and expansion state based on element equality
 - If elements remain stable, so will the view
 - Equality will work, identity creates less garbage
- When possible, debug elements should be reused across iterative suspends
 - Threads and stack frames representing the same frame in the same thread
 - Variables referencing and values representing the same objects

5. Debug Actions

- Actions operate on debug model elements
 - The step action works on instances of IStep, etc.
 - The actions update in response to selection change and debug events
- The standard debug actions are
 - Step over, Step into, Step return
 - Suspend, Resume
 - Terminate, Disconnect
 - Drop to frame

6. Debug Model Presentation

- Debug model elements are displayed with text and images

- Standard images are provided by the platform



- Default labels are just element names (for example, `IThead.getName()`)
- To provide custom labels and images
 - Contribute a `<debugModelPresentation>` extension
 - Provide corresponding implementation of `IDebugModelPresentation`, which is an `ILabelProvider`:

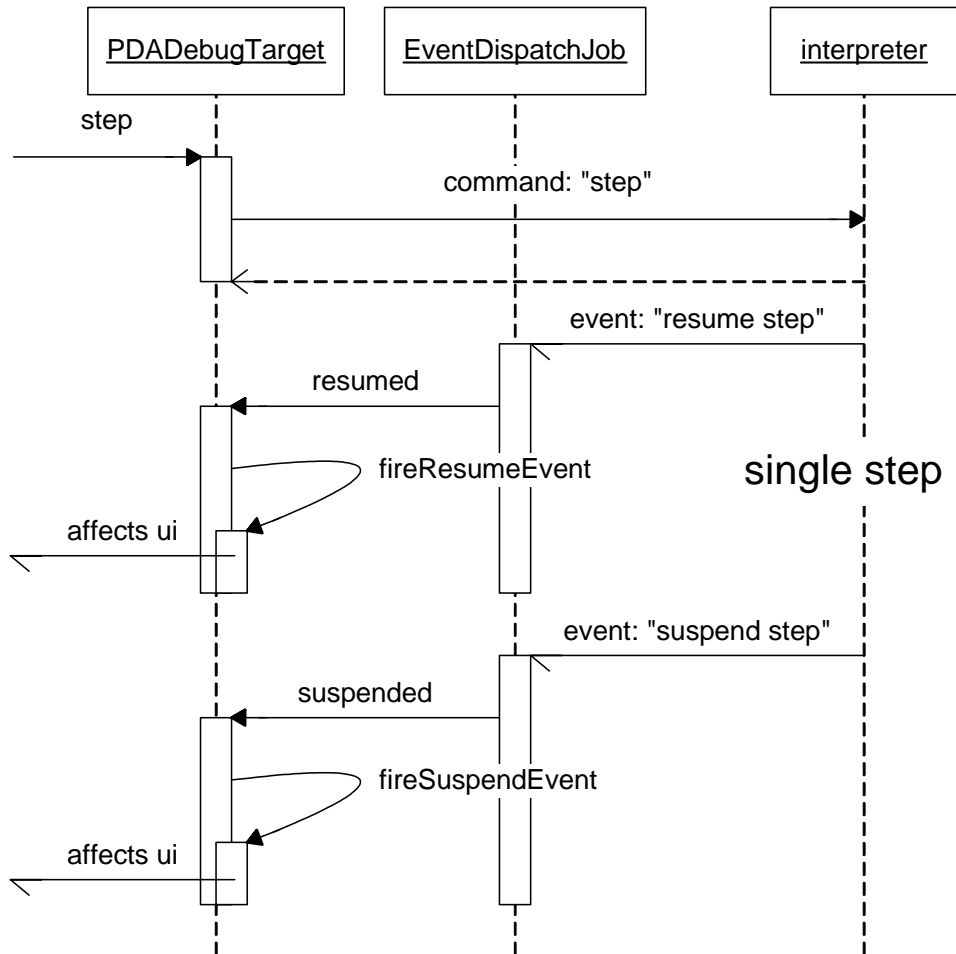
```
public Image getImage(Object element);
```

```
public String getText(Object element);
```

Example: Step Over

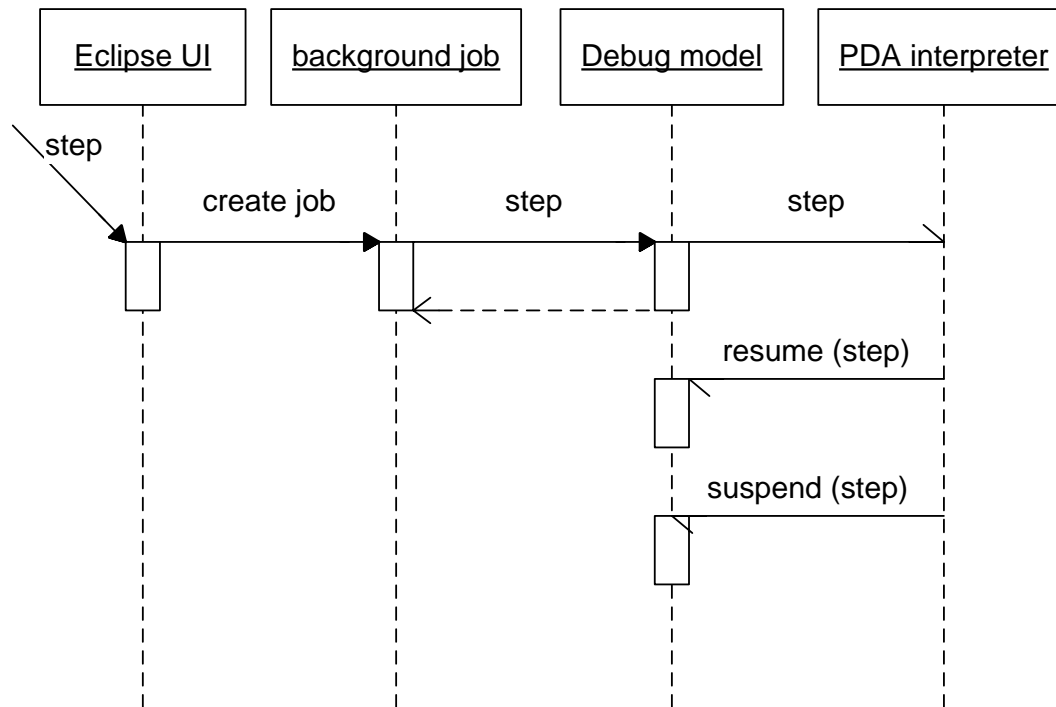
- The Eclipse platform is inherently multi-threaded thus all the classes must be thread safe
 - Explicitly (synchronization) or implicitly (by design)
- User interface actions, for example the step command, are issued from Eclipse's user interface thread.
 - Actions must be non-blocking.
 - We use step as the example here, because it might seem that step should block, that it should send the step debug command and wait for the interpreter to do the step, but that behavior would be wrong. Instead, the interpreter communicates with the debug model by firing events.

Step Over Continued



Step Over Continued

Even More Robust



Example: UI Updates for a Suspend Event

- A SUSPEND event fires due to a BREAKPOINT
 - Optionally switch perspectives and/or open the debug view
 - The debug view expands the suspended thread and selects its top frame
 - Selecting a stack frame triggers:
 - Source lookup, which adds instruction pointer to editor
 - Variables view refresh
 - Action enablement updates
- A SUSPEND event fires due to a STEP_END
 - The top stack frame is selected triggering source lookup, variables view refresh, and action enablement; thread and stack frame labels are refreshed
 - Perspective switch does **not** occur, which allows stepping in alternate perspectives

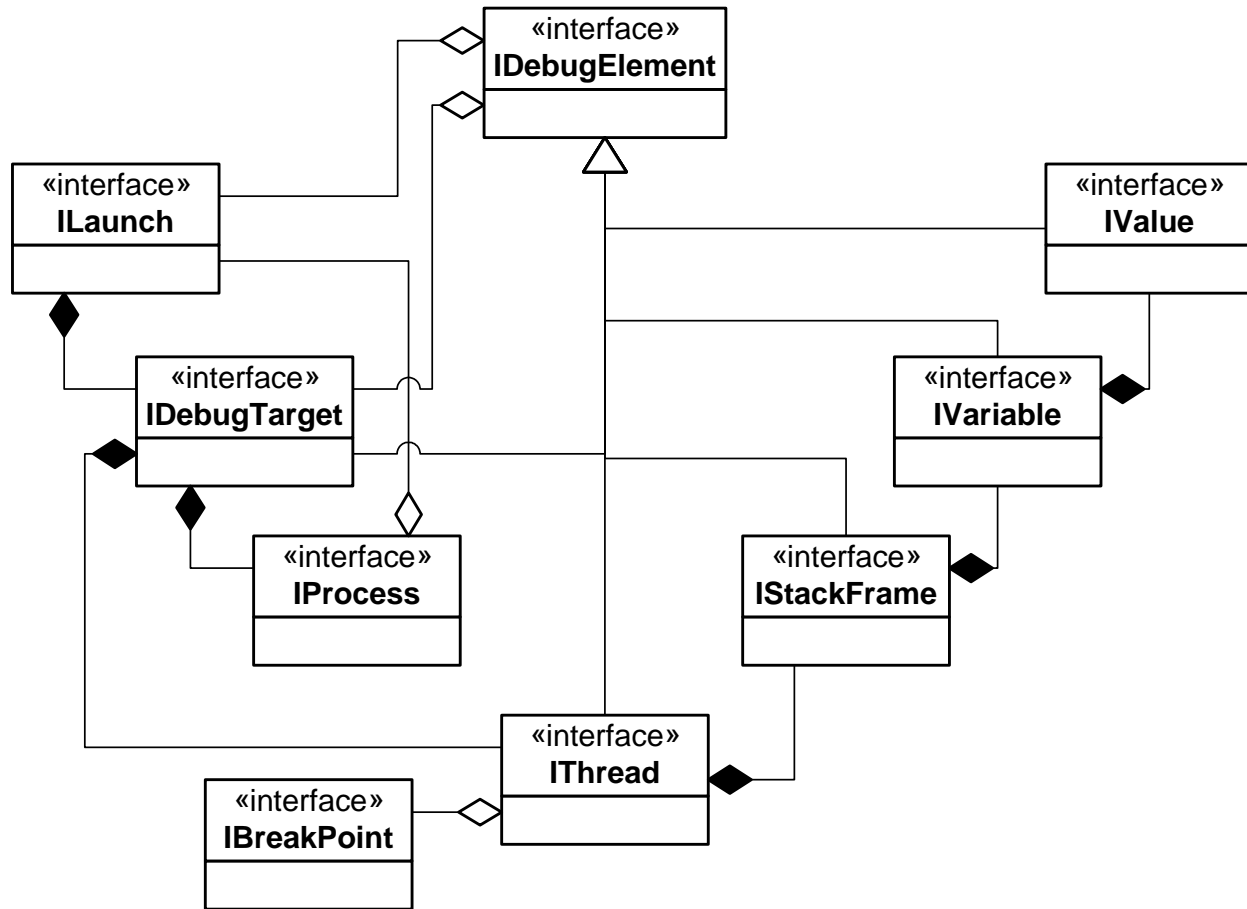
Example: UI Updates for a Resume Event

- A RESUME event fires with CLIENT_REQUEST
 - Debug view collapses the suspended thread
 - Selected stack frame becomes empty, clearing the variables view
 - Instruction pointers are cleared for the resumed thread
 - Actions enablement update
- A RESUME event fires with STEP_INTRO / OVER / RETURN
 - Thread and frames remain expanded/selected in the debug view
 - If a SUSPEND/STEP_END is not received within 500ms:
 - Thread label & icons update to show resumed (stepping)
 - Frames are cleared

Stepping & Breakpoints

- What happens when a step request encounters a breakpoint?
 - It depends on the underlying architecture
 - The Java debugger must clear its step request from the target VM when a breakpoint is encountered or else it will complete the step on the next request to resume
 - The Java debugger fires RESUME + STEP_OVER followed by a SUSPEND + BREAKPOINT event

The Debug Model



Checklist: Create Your Debug Model

1. Create implementations of **IDebugTarget**, **IThread**, **IStackFrame**, **IVariable**, and **IValue**
 - Also create classes to support communication with the actual target
2. If your target is not a standard operating system process, then also create an implementation of **IProcess**.
3. Instantiate your **IDebugTarget** in your **ILaunchConfigurationDelegate**

Exercise 2: Requests & Event Handling

- Goal:
 - Send resume, suspend, and terminate requests to the interpreter and handle the resulting events asynchronously
- We provide:
 - Plug-ins with stubbed out debug elements
 - PDA interpreter to accept requests and respond with events
- Your mission:
 - Code the resume, suspend and terminate request and event handling in the thread model element

Module 3

Breakpoints

Introduction

- What are breakpoints?
 - A way to suspend execution at a specified location or upon a specified condition.
 - Line breakpoints, Watchpoints, Run-to-Line, and Exception Traps are kinds of breakpoints.
- The breakpoint framework provides facilities for:
 - Add, remove, change notification
 - Persistence of breakpoints across workbench invocations
 - Temporarily skipping breakpoints
 - Retargetable actions for creating breakpoints
 - **New in 3.2:** Exporting and import breakpoints to/from a file

Introduction Continued

- The types of breakpoints a debugger provides depends on:
 - The capabilities provided by the underlying debug architecture
 - Aggregate functions that can be built with those capabilities
- Examples
 - Implementing run-to-line with line breakpoints
 - Implementing hit counts on the client side
 - Implementing conditional breakpoints with client side evaluations

Breakpoints Players

- Infrastructure
 1. Breakpoint extension – extension point for contributing kinds of breakpoints
 2. Breakpoint – model object representing an instance of a breakpoint
 3. Marker – used to persist a breakpoint’s attributes and display a breakpoint in an editor
 4. Breakpoint Manager – breakpoint repository, provides change notification
 5. Debug Target – installs breakpoints in underlying runtime

- UI
 6. Retargettable actions – actions for toggling breakpoints
 7. Editor – displays markers in vertical ruler
 8. Ruler Double Click – setting breakpoint action
 9. Breakpoint properties dialog – editing breakpoint properties

1. Breakpoint Extension Point

- The platform provides an extension point for contributing kinds of breakpoints

```
<extension point="org.eclipse.debug.core.breakpoints">
<breakpoint
  id="example.debug.core.pda.lineBreakpoint"
  name="PDA Line Breakpoint"
  class="example.debug.core.pda.PDALineBreakpointImpl"
  markerType="example.debug.core.pda.marker.lineBreakpoint"/>
</extension>
```

2. Breakpoint

- Model object representing a breakpoint
 - All breakpoints implement `IBreakpoint`
 - The platform also defines base interfaces for `ILineBreakpoint` and `IWatchpoint`
 - A breakpoint contains the information required to install itself into a debug target
 - All implementations must have a default constructor such that the platform can instantiate persisted breakpoints on workspace startup
 - The platform provides abstract classes that must be subclassed when implementing breakpoints – `Breakpoint` and `LineBreakpoint`.

2. Breakpoint Continued

- The platform's implementation provides enablement and attribute persistence
 - A breakpoint's attributes are stored in a marker
- Breakpoint behavior is provided by client implementations
 - I.e. Custom properties, installation, adhering to enablement
- For example, the Java debugger supports:
 - Line breakpoints, watchpoints, method breakpoints, exception breakpoints, class load breakpoints
 - Hit counts, suspend VM vs. thread, conditions, thread filters, location filters, instance breakpoints, suspend on caught vs. uncaught
 - An API for creating and configuring these breakpoints
 - Property pages and actions for editing Java breakpoint properties

2. Continued: Breakpoint Mechanisms

- The actual mechanism used to suspend execution on a debuggable process differs between debug architectures
 - E.g. JVM debug interface accepts requests to suspend execution at specific line number in a class file
 - This only works when line number debug attributes are present in the class file
- An **IBreakpoint** simply stores the information required to install a breakpoint in a specific architecture
 - For example, a Java line breakpoint stores a fully qualified type name and line number.
 - When added to a target, we first check if the corresponding class is loaded, and if so make a request to suspend at the associated location. If not loaded, we ask to be notified when the class is loaded and install the request later (deferred breakpoint).

3. Marker

- Breakpoint attributes are stored in markers
 - `IMarker`'s are provided by the platform as general purpose “markers” in files (book marks, compilation errors, etc), that can be displayed in an editor ruler
 - An marker is just a store of key/value pairs of primitive data types
 - The platform provides the only implementation of `IMarker`
- Breakpoint behaviors are implemented in `IBreakpoint`'s
 - To provide complex breakpoint behavior, debuggers provide implementations of `IBreakpoint`
 - All breakpoints have an associated marker to persist it attributes and display in an editor

3. Marker Continued

- Contribute marker extension associated with breakpoint type

```
<extension
  id="pda.marker.lineBreakpoint"
  name="PDA Line Breakpoint Marker"
  point="org.eclipse.core.resources.markers">
  <super type=
    "org.eclipse.debug.core.lineBreakpointMarker" />
  <persistent value="true" />
</extension>
```

- Notes
 - Must specify persistent as true if you want your breakpoints to be persisted
 - Must specify attributes you want persisted

4. Breakpoint Manager

- The breakpoint manager (`IBreakpointManager`) is a repository of breakpoints in the workspace
 - When a breakpoint is created, it is registered with the manager
 - When a breakpoint is deleted, it is removed from the manager
 - Provides change notification as breakpoints are added, removed, and when a breakpoint attribute changes
- Clients interested in breakpoints implement `IBreakpointsListener` and register with the manager for change notification
 - For example, debug targets listen for change notification so they can install/remove/update breakpoints as they change
- Clients should also register as `IBreakpointManagerListener`'s
 - Notified when the breakpoint manager has been disabled/enabled
 - This feature allows all breakpoints to be temporarily disabled with out changing the enablement state of individual breakpoints (i.e. skip breakpoints)

5. Debug Target

- The debug target installs breakpoints
 - When a debug target is created, it should query the breakpoint manager for all existing relevant breakpoints and install them (deferred breakpoints)
 - Listens for breakpoints being added/removed/changed during its lifecycle, and updates them in the underlying runtime

6. Retargetable Actions

- Most debuggers support a common set of breakpoint types
 - Line breakpoints, method breakpoints, and watchpoints
- Global actions are provided for creating these kinds of breakpoints
 - Promotes a common look and feel across debuggers
 - Avoids polluting menus with similarly named actions
- How does it work?
 - The global actions ask the active editor or view for its `IToggleBreakpointsTarget` adapter. Debuggers register adapters with the appropriate views and editors.
 - Adapter can be queried to determine action enablement on the current selection (text or otherwise) – `canToggleLineBreakpoint()`
 - Adapter delegated to toggle breakpoints – `toggleLineBreakpoints()`

7. The Editor

- The editor visualizes the location of breakpoint/watchpoints
 - Displays markers in vertical ruler and updates as markers are changed
 - Provides adapter to hook into the retargetable breakpoint actions
 - Editors that subclass `AbstractDecoratedTextEditor` have a ruler to display markers associated with the file (resource) they are editing

8. Ruler Double Click

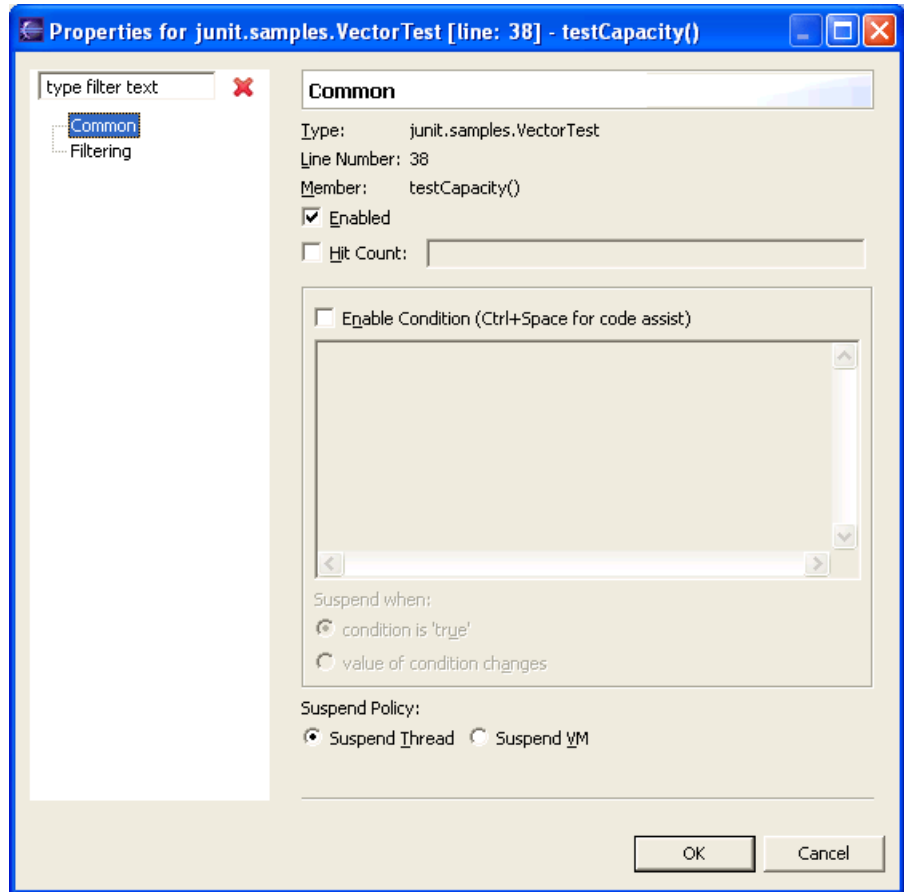
- Double click in ruler can have more than one action depending on context
 - Set a line breakpoint (line or method entry)
 - Set a watchpoint
- Hooking up a double-click action in an editor's ruler
 - Use workbench extension point to contribute an editor action
 - Refers to an **AbstractRulerActionDelegate** that provides an action to do the desired work
 - Added via the `<editorActions>` extension point

8. Ruler Double Click Continued

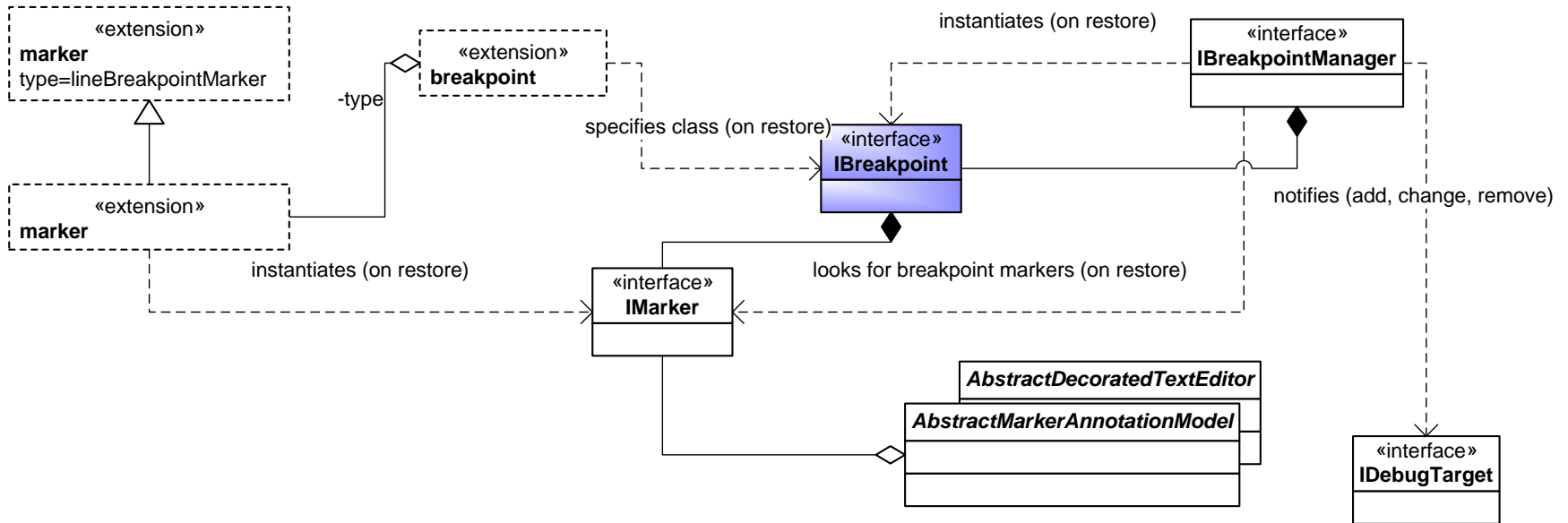
- There are two levels of indirection in the breakpoint double-click infrastructure:
 - The first level is part of the text editor infrastructure
 - A contributed **RulerDoubleClick** action creates a delegate action to perform the actual double click action
 - In our case, the **RulerToggleBreakpointActionDelegate** creates a **ToggleBreakpointAction**
 - The second level is part of the debug framework
 - The debug platform provides a standard action that you can contribute as a **RulerDoubleClick** action
 - Its delegate interacts with the standard **IToggleBreakpointsTarget**
 - With an **IToggleBreakpointsTarget** adapter, you can contribute the standard **RulerToggleBreakpointActionDelegate** to an editor.

9. Breakpoint/Watchpoint Properties

- Breakpoints have editable properties
 - Hit count
 - Suspend policy
 - Enablement
 - Condition
- Breakpoint editors are not currently provided by the platform
 - The Java debugger provides its own



The Breakpoint Model



Checklist: Breakpoints

1. Determine the kinds of breakpoints your debugger will support and contribute an `org.eclipse.debug.core.breakpoint` extension for each
2. Create an implementation of `IBreakpoint` for each
3. Define the `org.eclipse.core.resource.marker` extension associated with each breakpoint kind, enumerating attributes
4. Implement breakpoint installation in your debug target via its `IBreakpointListener` interface
5. Implement `IBreakpointManagerListener` in your debug target to support “skip breakpoints”
6. Ensure that you have a source code editor; this can be as simple as a subclass of the standard `TextEditor`
7. Create an implementation of an `IToggleBreakpointsTarget` adapter using, and register the adapter with your editor
8. Contribute a `RulerToggleBreakpointActionDelegate` to your editor using the `<editorActions>` extension point
9. Create breakpoint properties editor if desired

Exercise 3: Create & Install Breakpoints

- Goal:
 - To understand the breakpoint lifecycle: creation, installation, modification, and deletion.
- We provide:
 - Plug-ins with stubbed out actions, breakpoint, and editor implementation
- Your mission:
 - Define a line breakpoint extension in plug-in XML and its associated marker extension
 - Code installation and removal of line breakpoints

Module 4

Source Lookup

Introduction

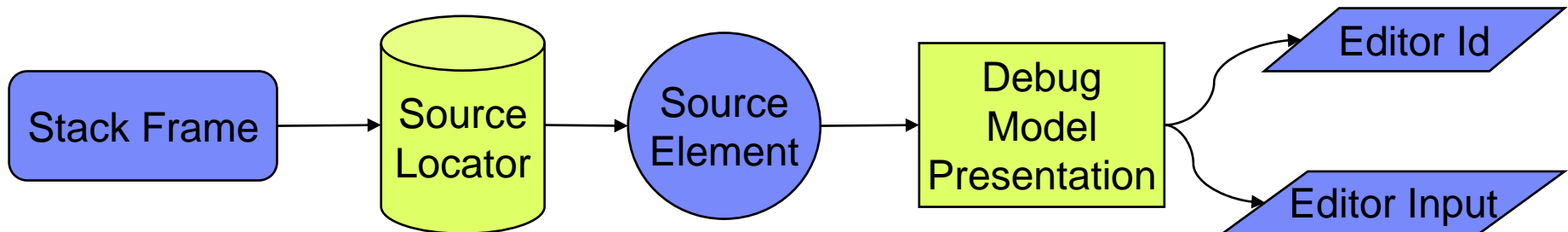
- Highlighting the current source code line or statement is mandatory in modern debuggers
- The debugger has to find source code for a binary location and display that source code in the editor.
- Typically, finding source code means looking for a particular file along a path of directories (or zips or jars or databases or ...)

Basic Source Lookup Players

- Infrastructure
 - Launch – each launch has a source locator
 - Source Locator – locates source element for a stack frame
 - Stack Frame – provides context for source lookup
- UI
 - Debug Model Presentation – provides editor mapping for source element

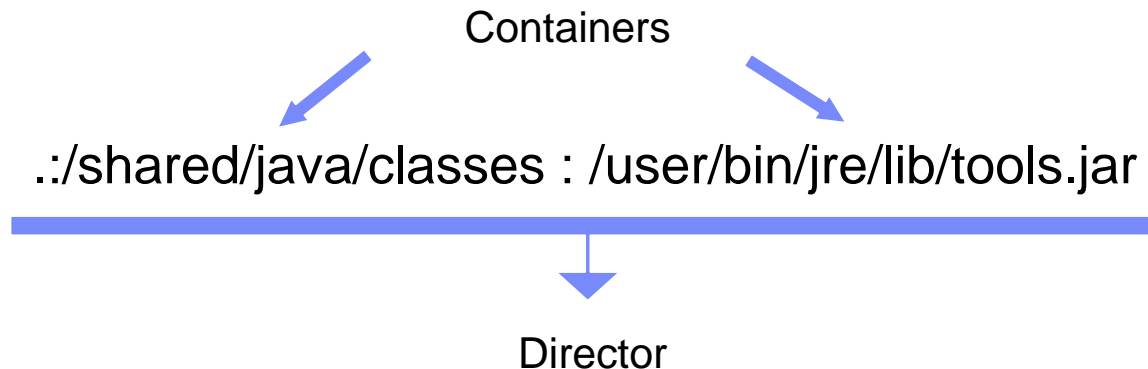
The Basic Source Lookup Interaction

1. A stack frame is selected
2. The source locator finds a *source element* for a stack frame
3. The debug model presentation maps the source element to an *editor input* and *editor id*
4. The platform opens the editor
 - Positions to the line specified by the stack frame
 - Adds instruction pointer annotation for the stack frame



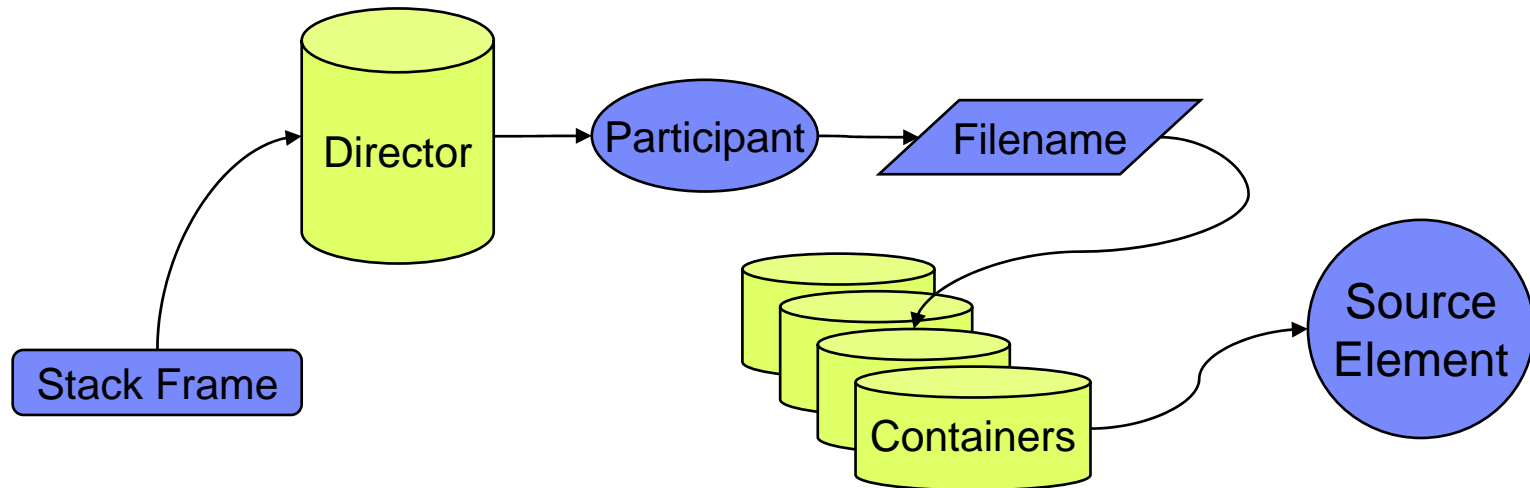
Source Lookup Framework

- An implementation of a source locator with a standard ‘search along a path’ type of source locator, which consists of:
 - director – holds the ordered list that is the “path”
 - participants – map stack frames to filenames
 - containers – find files by filename in directories, zips, jars, etc.
- The director-container-participant trio is similar to a Unix path



Director Interaction

- To locate source for a stack frame, the director iterates through its participants. For each participant:
 1. The director asks the participant to translate the stack frame into a source file name
 2. The director iterates through its source containers asking each container for the source element matching a file name



Source Lookup Director Continued

- The default implementation of the source lookup director provides an implementation of a 'path' as a consistently ordered sequence of containers
- Leaving only two artifacts to be created
 - Initial participants – the objects that map stack frames to file names
 - Initial set of containers – the places to search for source files
- If the user has not specified an explicit source path, the director computes a default source path (set of source containers), based on launch configuration type.

Source Lookup Participants

- A source lookup director usually has one participant
 - We allow for multiple participants to support multi-language debug scenarios where there can be different kinds of stack frames requiring participants from each model to provide source file names
- To initialize your director with participants
 - Subclass `AbstractSourceLookupDirector` and override `initializeParticipants()`

```
public void initializeParticipants() {  
    addParticipants(new ISourceLookupParticipant[] {  
        new JavaSourceLookupParticipant()});  
}
```

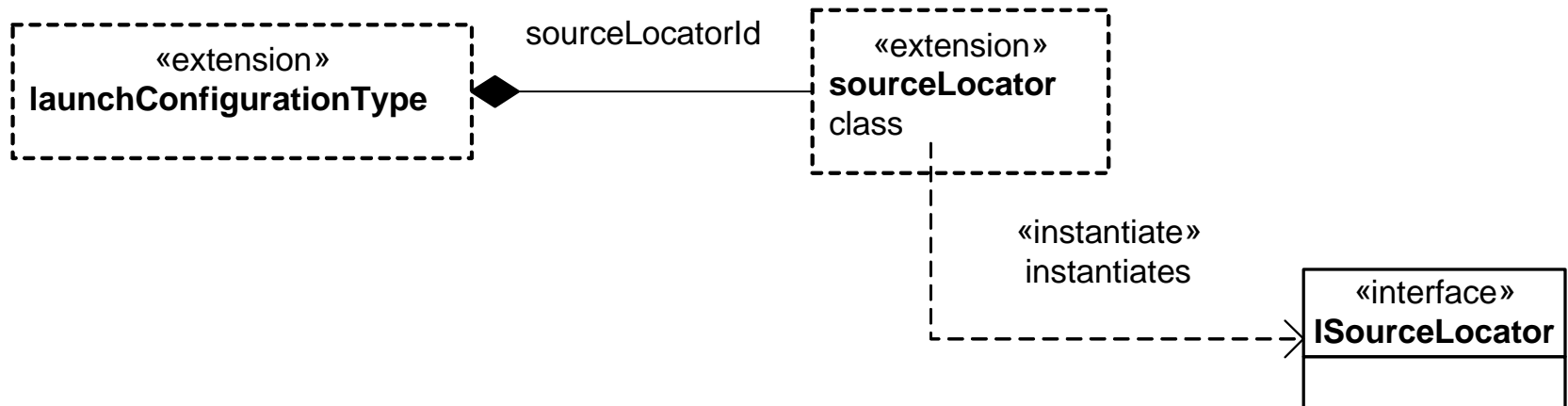
Default Source Lookup Path

- The default source lookup path consists of one container
 - the `DefaultSourceContainer`
 - uses an `ISourcePathComputer` to compute the default source path.
- On each launch the source path computer computes the source path
 - Allows a source path to be dynamically generated on each launch
- Use the `<sourcePathComputer>` extension to contribute a source path computer.
 - A launch configuration type extension can specify a source path computer
 - You implement `ISourcePathComputerDelegate` which computes a default set of source containers for a launch configuration

```
public ISourceContainer[] computeSourceContainers(  
    ILaunchConfiguration, IProgressMonitor);
```

Instantiating a Source Locator

- A launch delegate can instantiate a source locator and set it on a launch object
- However, if you don't want to write code to instantiate the source locator when your launch delegate sets up the launch, then...
 - The framework can use the `<launchConfigurationType>` and `<sourceLocator>` extensions to instantiate the source locator for you

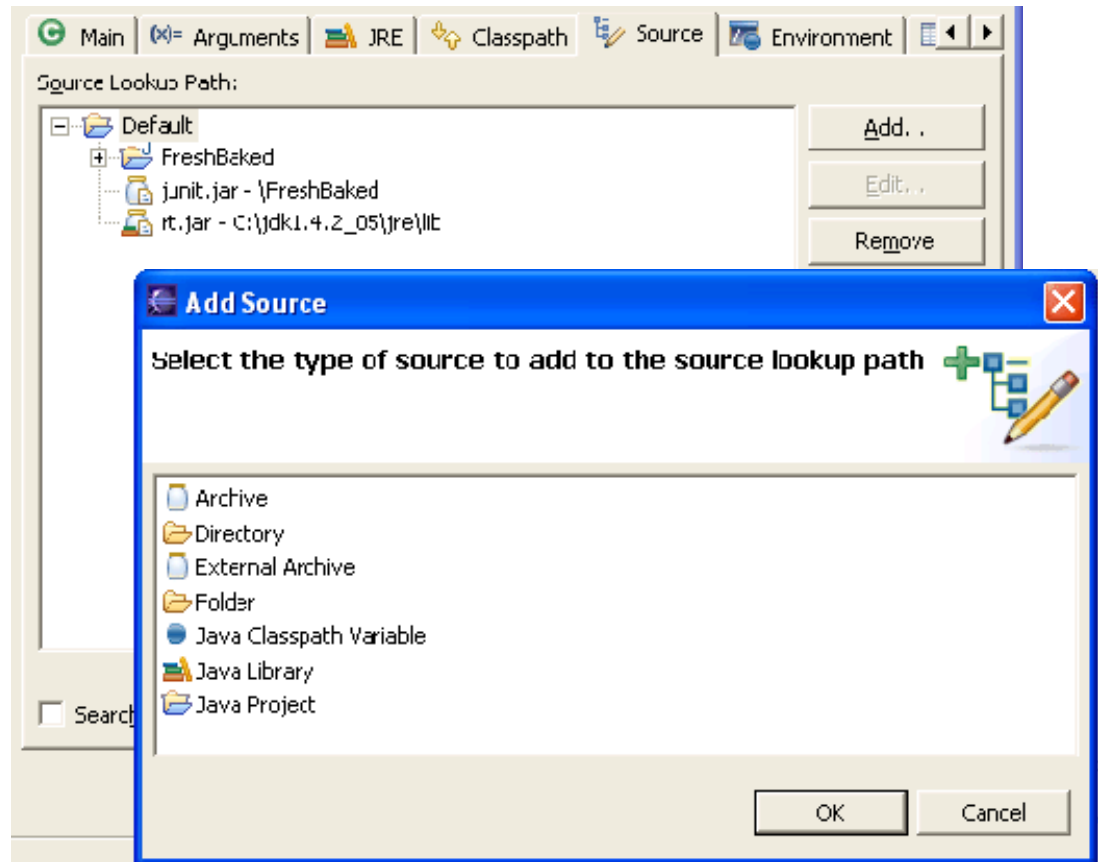


Source Containers

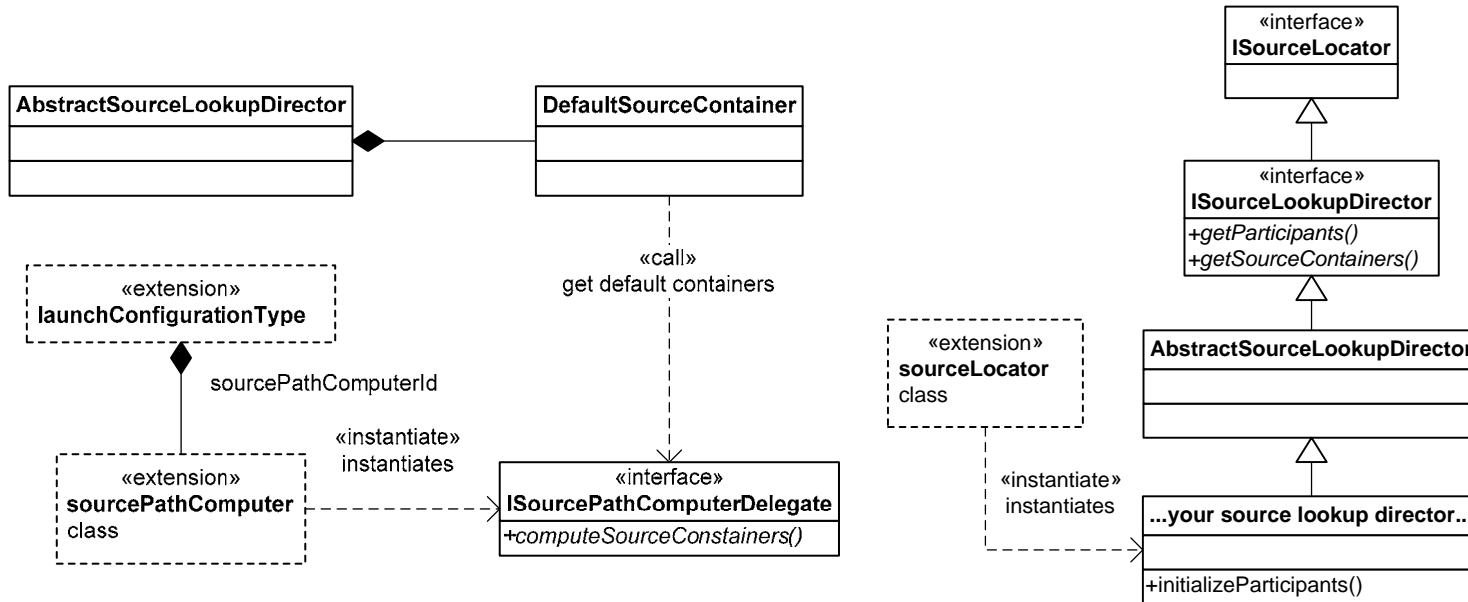
- The platform provides implementations of standard source containers
 - Workspace folders, projects, and archives
 - Local file system directories and archives
- The set of containers is extensible
 - Contribute new types of containers with the `<sourceContainerTypes>` extension point
 - If you contribute a source container, you also need to contribute a corresponding `<sourceContainerPresentation>` to provide an icon and browser
 - The browser is used to choose and edit a source container

Source Path Editing

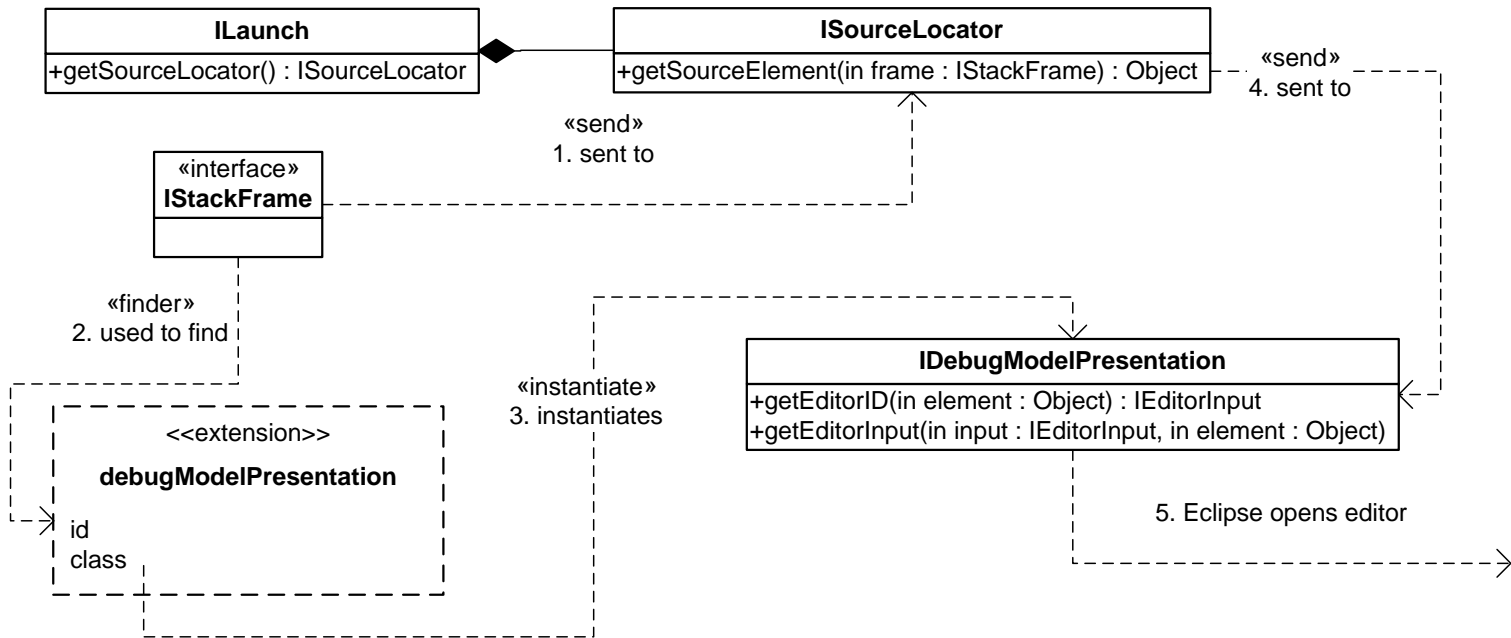
- The framework provides a UI to edit a source lookup path
 - Source lookup tab – launch configuration tab provides a UI for configuring and modifying a source lookup path
 - Users may specify an explicit path (set of containers) to search, or specify to use the default source path



The Source Lookup Model



The Source Lookup Model Continued



Checklist: Source Lookup Hookup

1. Create an implementation of `ISourceLookupParticipant` to provide file names for your stack frames
2. Create subclass of `AbstractSourceLookupDirector` and override `initializeParticipants` to instantiate your participant (1)
3. Create implementation `ISourcePathComputerDelegate` that computes a default source lookup path for your launch configurations
4. Define the `<sourceLocator>` extension to point to your director (2)
5. Define the `<sourcePathComputer>` extension to point to computer (3)
6. In your `<launchConfiguration>` extension, refer to (4) and (5)

Exercise 4: Add Source Lookup

- Goal:
 - Learn how the source lookup players are connected to a launch.
- We provide:
 - Stubbed out source lookup director, participant, and source path computer.
- Your Mission:
 - Define plug-in XML to hook the PDA launch configuration type to the PDA source lookup director and source path computer
 - Code initialization of the source lookup director to add its participant
 - Code participant to translate a stack frame to file name
 - Code source path computer to provide default source lookup path

Module 5

Variable Customizations

Introduction

- The Variables View provides facilities for:
 - Emphasizing variables that change value
 - Displaying “logical structures” vs. raw implementation structures
 - Displaying “details” (`toString()`) for a selected variable
 - Installing context sensitive code assist in the details area
 - Editing variable values

1. Emphasizing Variable Value Changes

- Variables that change value while stepping are rendered in red
 - A variable must implement `hasValueChanged()` for this to work
 - Spec says: “returns whether the value of this variable has changed since the last suspend event”
- Implementation in the Java debugger
 - The debug target maintains a count of how many times it has suspended; the count is incremented each time a thread suspends representing arbitrary points in time: 0, 1, 2, ...
 - Each time a variable retrieves its value, it checks if it has changed since the last time it was asked, and if so, synchronizes its counter with the debug targets suspend count.
 - When a variable counter == debug counter it's value has changed

2. Logical Structures

- A logical structure is an alternate presentation of a variable's value
 - Often, it's more natural to navigate a complex data structure via an alternate semantic presentation of the value, rather than its implementation.
 - For example, no matter how a list is implemented (linked, array, etc.), a user wants to see the elements in the list in terms of an ordered collection
- There are two extension points for contributing logical structures
 - The `<logicalStructureTypes>` extension point is used to contribute one logical structure per variable value
 - The `<logicalStructureProviders>` extension point is used to contribute a delegate that can provide multiple logical structures for a variable value dynamically

Logical Structure Type Extension

- Provides a single logical structure for per value
 - Has a description that is presented in the “Show As >” menu
 - For example, “Show As > Array”
 - Contributes an `ILogicalStructureTypeDelegate` to translate value

```
public boolean providesLogicalStructure(IValue)
public IValue getLogicalStructure(IValue)
public String getDescription(IValue)**
```

** (this method is actually defined by

```
ILogicalStructureTypeDelegate2)
```

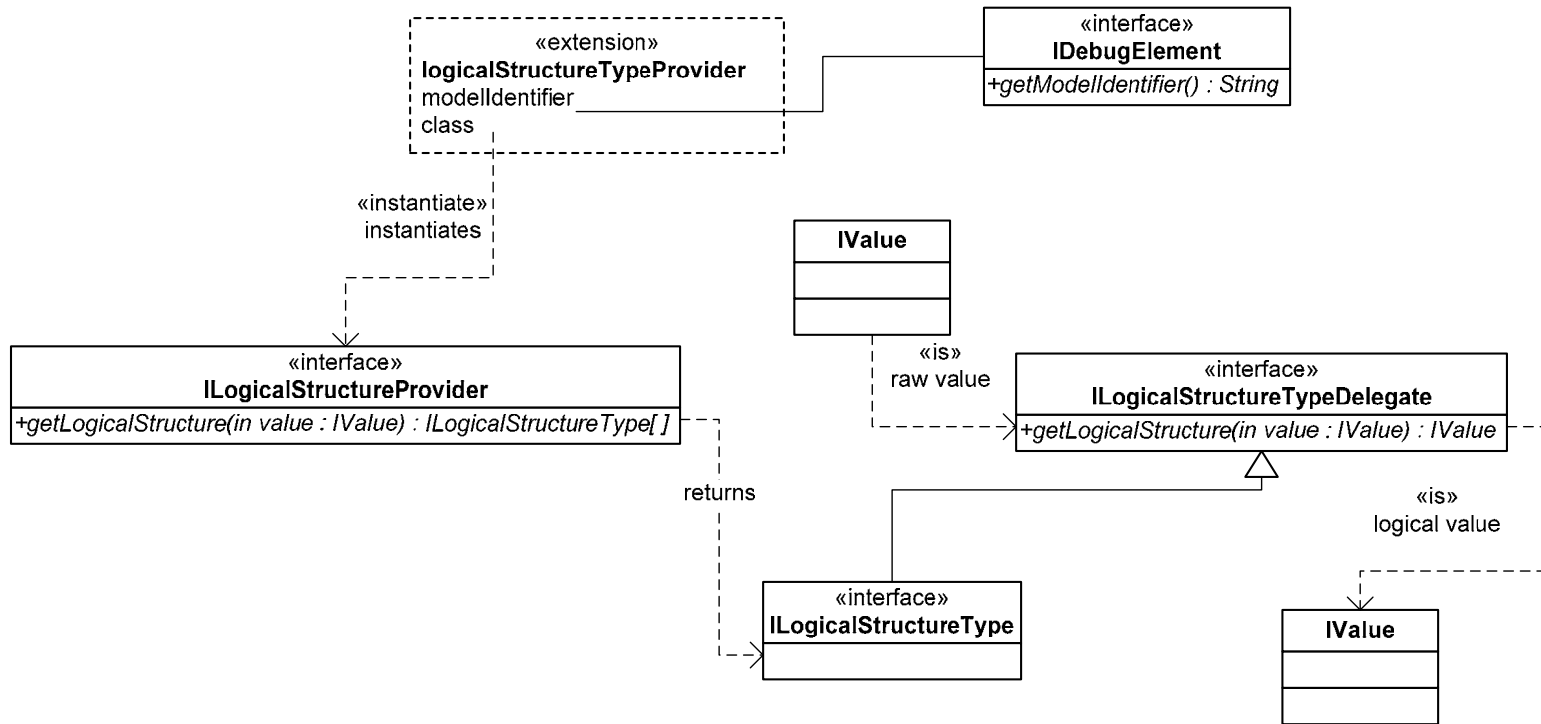
Logical Structure Provider

- Provides logical structure types for a variable value by implementing `ILogicalStructureProvider`

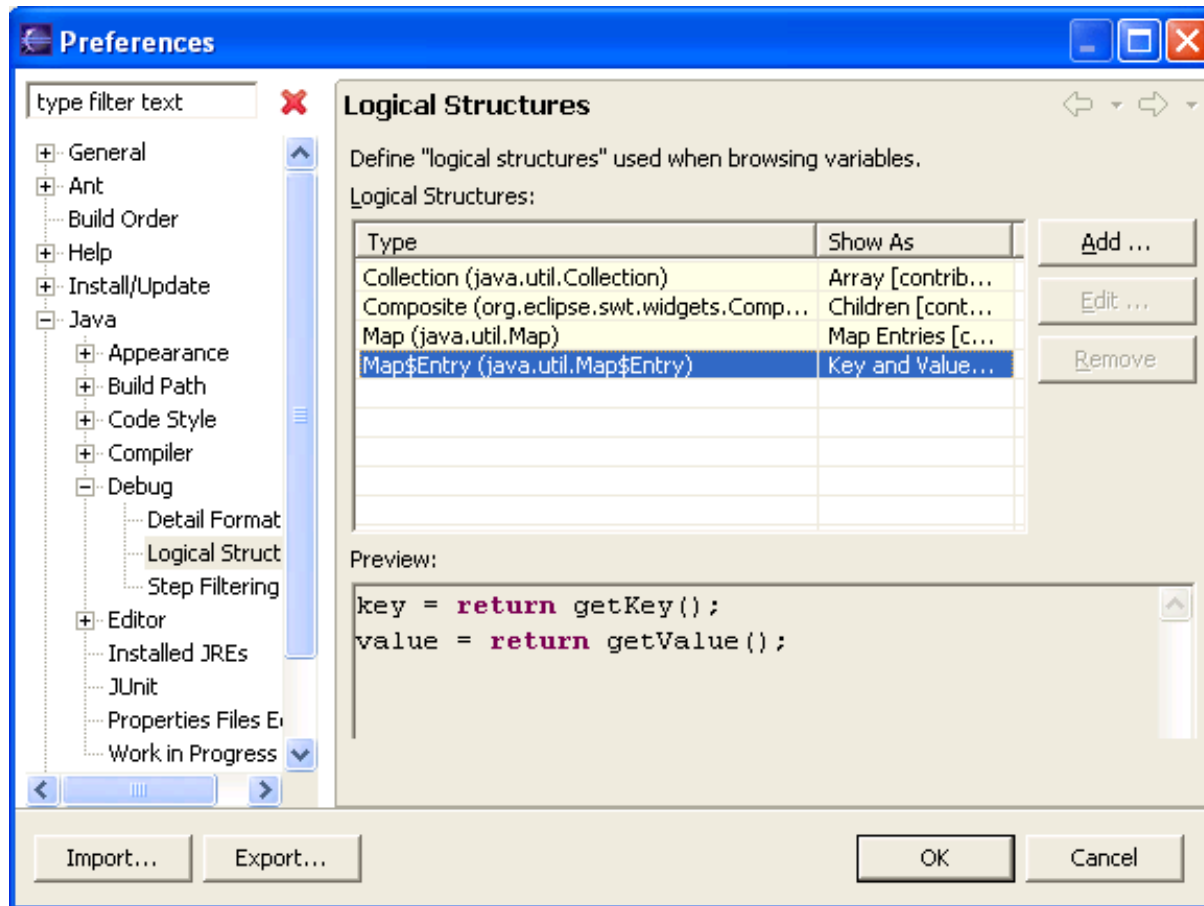
```
public ILogicalStructureType[]  
    getLogicalStructureTypes(IValue)
```

- Allows for a dynamic set of structures to be provided per value
- For example the Java debugger uses this extension point to allow users to configure logical structures via code snippets

2. Logical Structures Continued

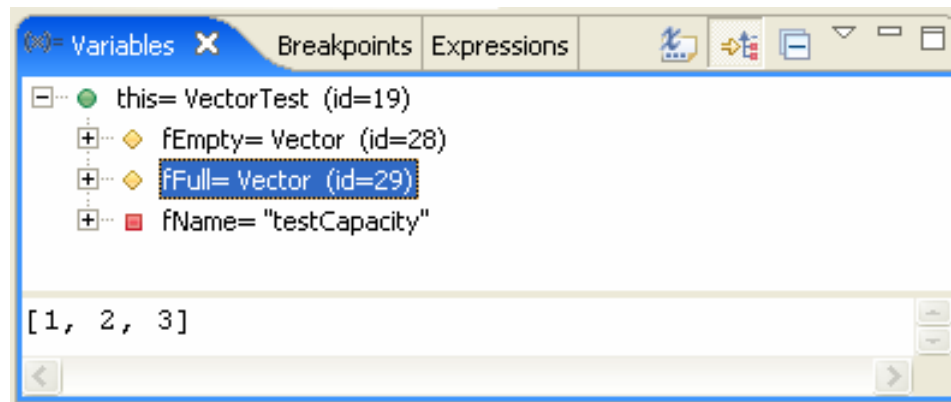


2. Logical Structures Continued



3. Variable Details & Code Assist

- The details area of the variables view displays extra information for a selected variable.
 - Text is provided by a debug model presentation asynchronously
 - A debug model presentation can specify a `SourceViewerConfiguration` to install in the details area
 - Allows for code assist, formatting, etc.



4. Variable Editing

- The variables view provides a standard dialog for editing a variable's value (for variables that support value modification)
 - You can also type into the details area and save (Ctrl-S)
- The `<variableValueEditor>` extension point allows debuggers to provide specialized dialogs and to handle the save (Ctrl-S) operation
 - Delegated to when editing a variable or when save is invoked
 - The Java debugger provides specialized dialogs for primitives, Strings, and Objects
 - The Java debugger handles the save operation by performing an evaluation on the selected text to compute a new value for the selected variable

Checklist: Customizing Variables

1. Implement `IVariable`'s `hasVariableChanged(...)` so variables turn red when they change value
2. Implement an `ILogicalStructureType` and an `ILogicalStructureProvider` and contribute an `<logicalStructureProvider>` extension to provide logical structures
3. Contribute a `<variableValueEditor>` extension to override the default value editor provided by the platform

Exercise 5: Logical Structures

- Goal:
 - Contribute a logical structure that displays the words in a value
- We provide:
 - A stubbed out logical structure delegate
- Your mission:
 - Define plug-in XML to contribute the logical structure
 - Code the translation from a value to its words

Module 6

Custom Views and Actions

Introduction

- The debug platform provides a single debug perspective and set of views and actions generally useful to most debuggers
 - One might call it a “general weak” solution
- Clients developing IDEs want to build highly integrated tools with IDE specific features
 - They want a “strong specific” solution

Customizing the Debug Perspective

A debugger can customize the debug perspective by contributing to standard workbench extension points to:

1. Contribute custom actions
2. Contribute custom views
3. Define and filter product capabilities

As well, a debugger can use the debug platform's view binding extension point to control custom views:

4. Control when custom debug views open and close

Sharing the Debug Perspective

- We encourage tool developers to re-use the debug perspective for their debuggers
 - Provides a common look and feel for all debuggers
- Problem: the perspective may become polluted with many tool specific views and actions
 - Use action “object contributions” specific to your debug elements
 - Used the platform’s console view when implementing a console
 - Use workbench “capabilities” to filter views and actions

1. Custom Actions: Objects & Views

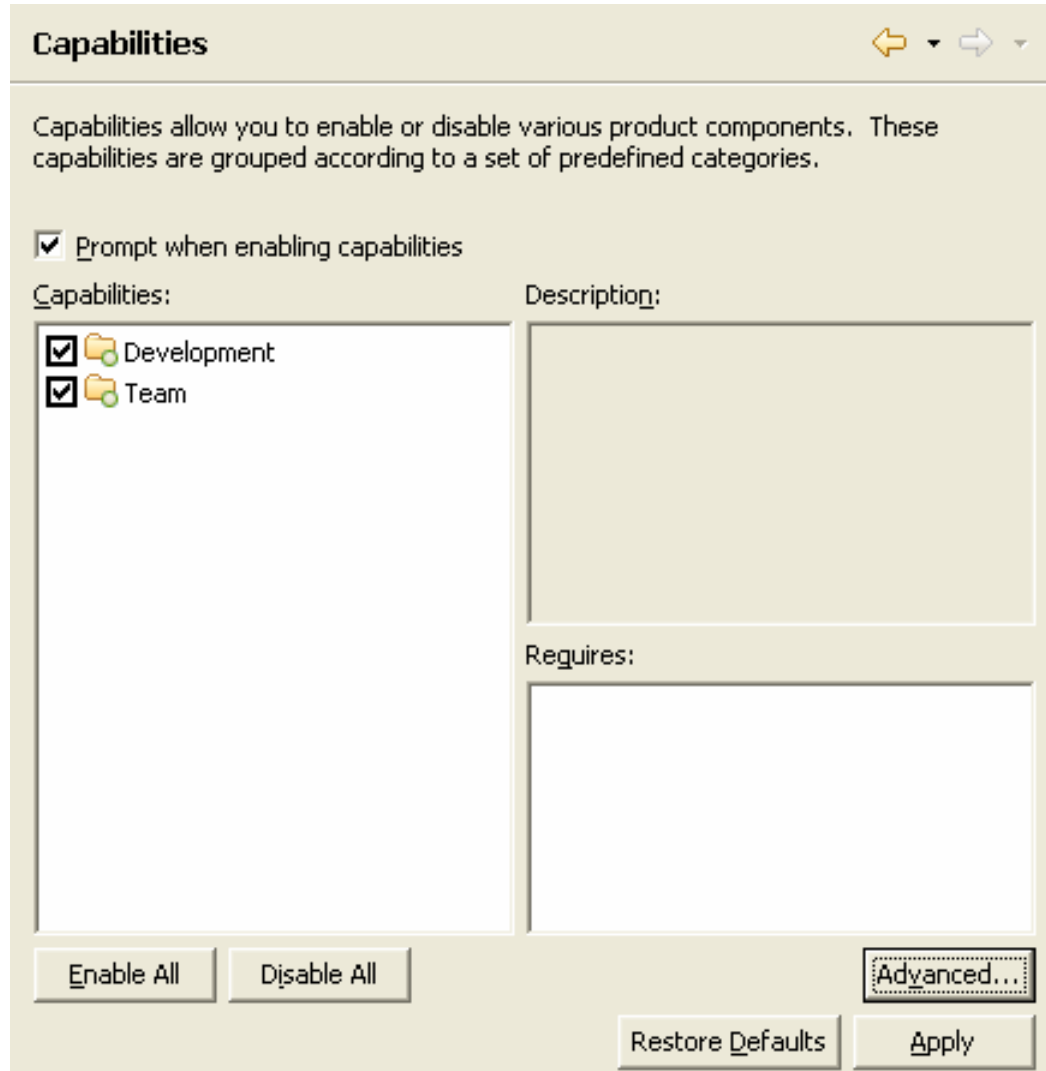
1. To contribute an action to a specific kind of object, use an object contribution (extension point `<org.eclipse.ui.popupMenus>` with an `<objectContribution>` element)
 - For example, the Java debugger contributes actions to the debug view context menu to open the declaring and receiving type hierarchies of the selected stack frame. The action is contributed for instances of `IJavaStackFrame`.
2. To contribute an action to a specific view use the viewer contribution (extension point `<org.eclipse.ui.viewActions>`, with an `<action>` element)
 - For example, the Java debugger contributes actions to the debug view drop-down menu to show system threads, monitors, etc.

2. Custom Views

- A debugger can contribute custom views to the debug perspective by using the `<org.eclipse.ui.views>` extension point
 - The view should be contributed to the `org.eclipse.debug.ui` category so it appears in the Show View menu with other debug views
 - For example, the Java debugger contributes the Display view to support expression evaluations
- A debugger can provide a default position a custom view in the debug perspective by using the `<org.eclipse.ui.perspectiveExtensions>` extension point with a `view` element
 - The view can be positioned relative to other views in the perspective
 - For example, the Display view is stacked with the console view

3. Capabilities

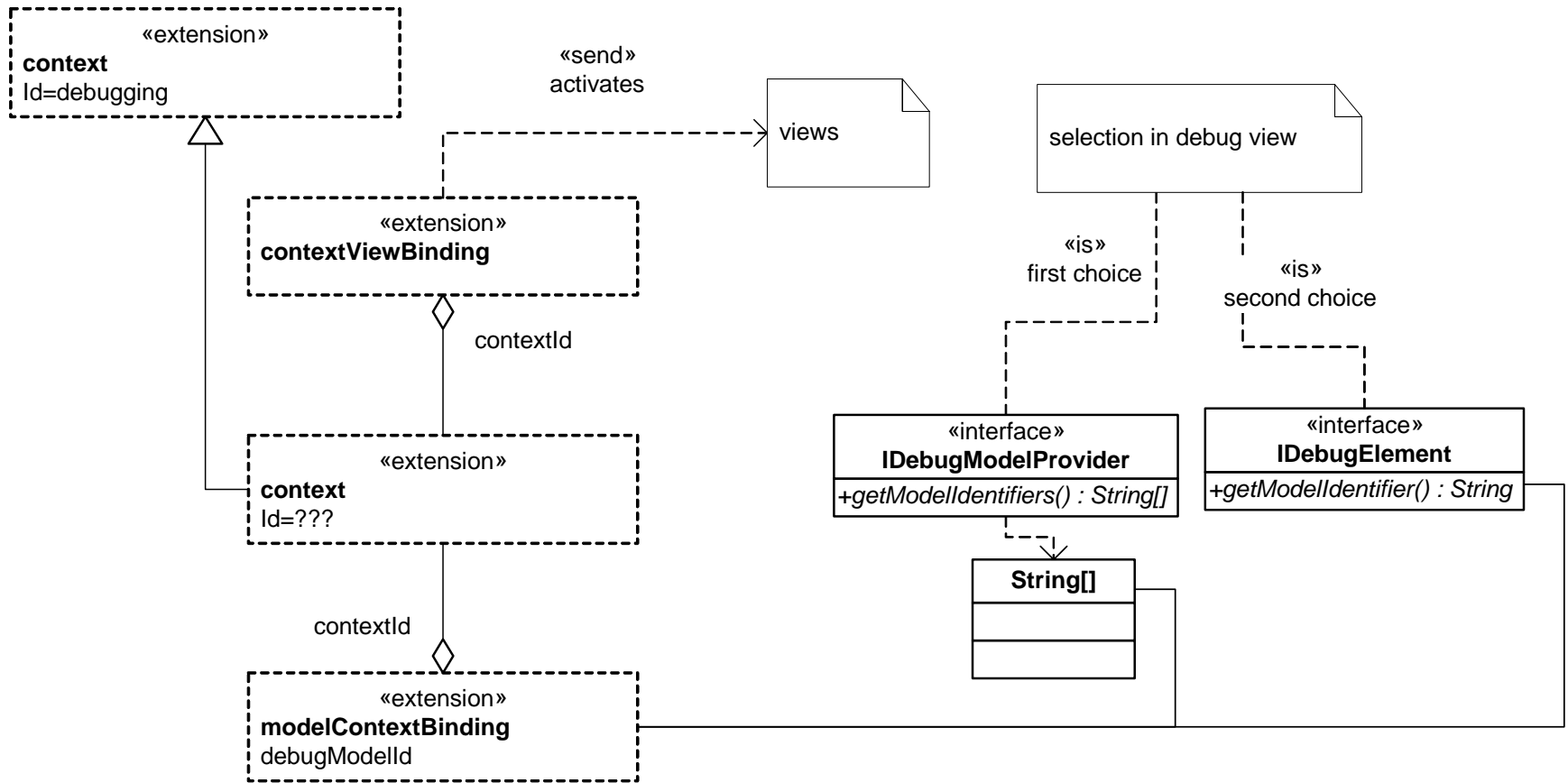
- The SDK distinguishes between JDT, PDE, and CVS
- When a capability is off, its contributions (views and actions) are not visible in the workbench
 - Capabilities have trigger points – for example, creating a project
- Best practice: product manager creates a separate plug-in to define the <activity> extensions that define the capabilities



4. Automatic View Open & Close

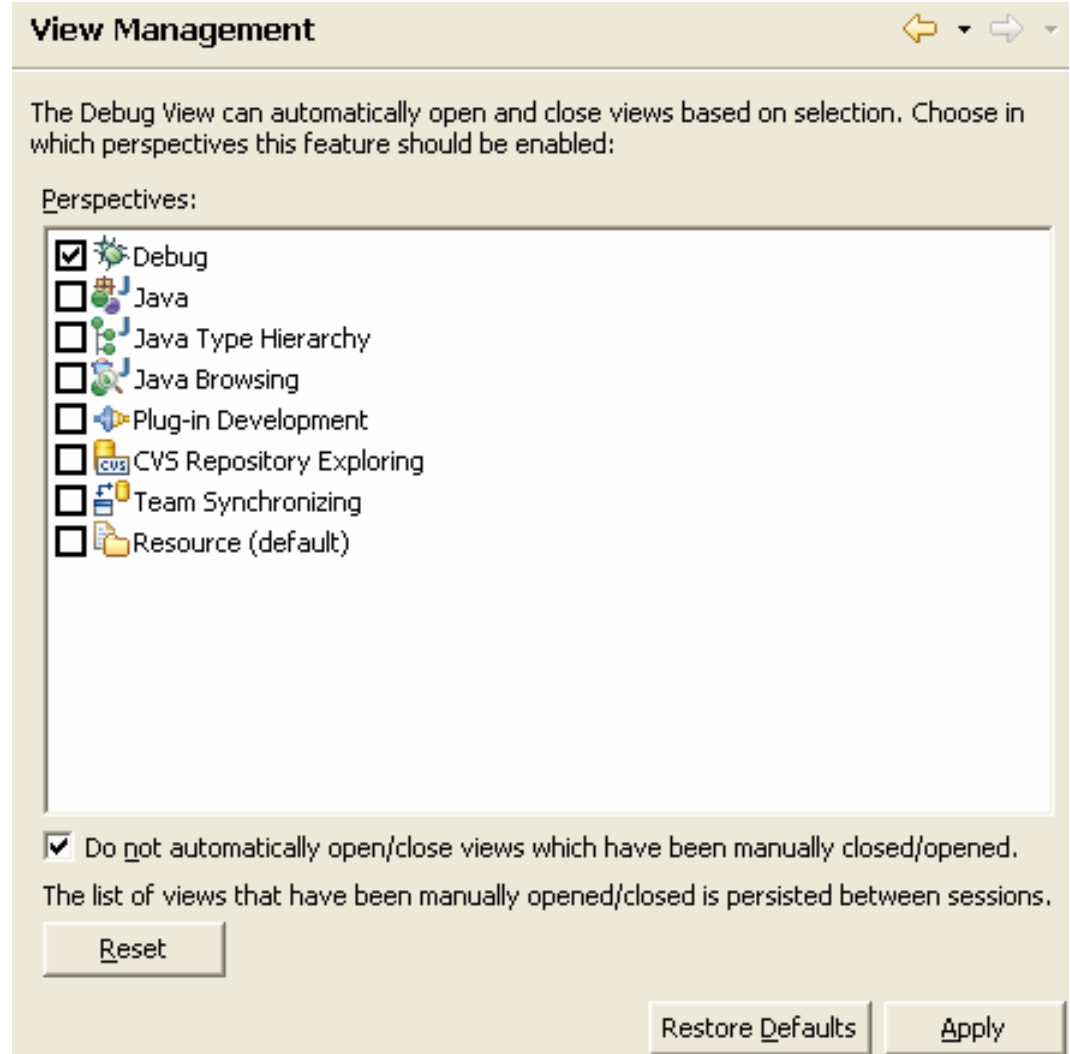
- The debug platform provides support to automatically open views as stack frames from different debuggers are selected
 - The views can be automatically closed when a debug session terminates
- Use extension points to
 - Bind your debug model to a workbench context
 - Bind views to a workbench context specifying auto open/close
 - Contribute your debugger's context (which can inherit from the debug platform's debug context)

4. View Bindings Continued



4. View Management

- Users can control the perspectives in which debugging is available



Checklist: Custom Views & Actions

1. Add custom menu items with object contributions and/or view contributions
2. Add custom views with the views extension and position them with the perspective extension
3. Define product capabilities
4. Contribute view and debug model bindings to control view opening and closing

Module 7

Editor-Debugger Interactions

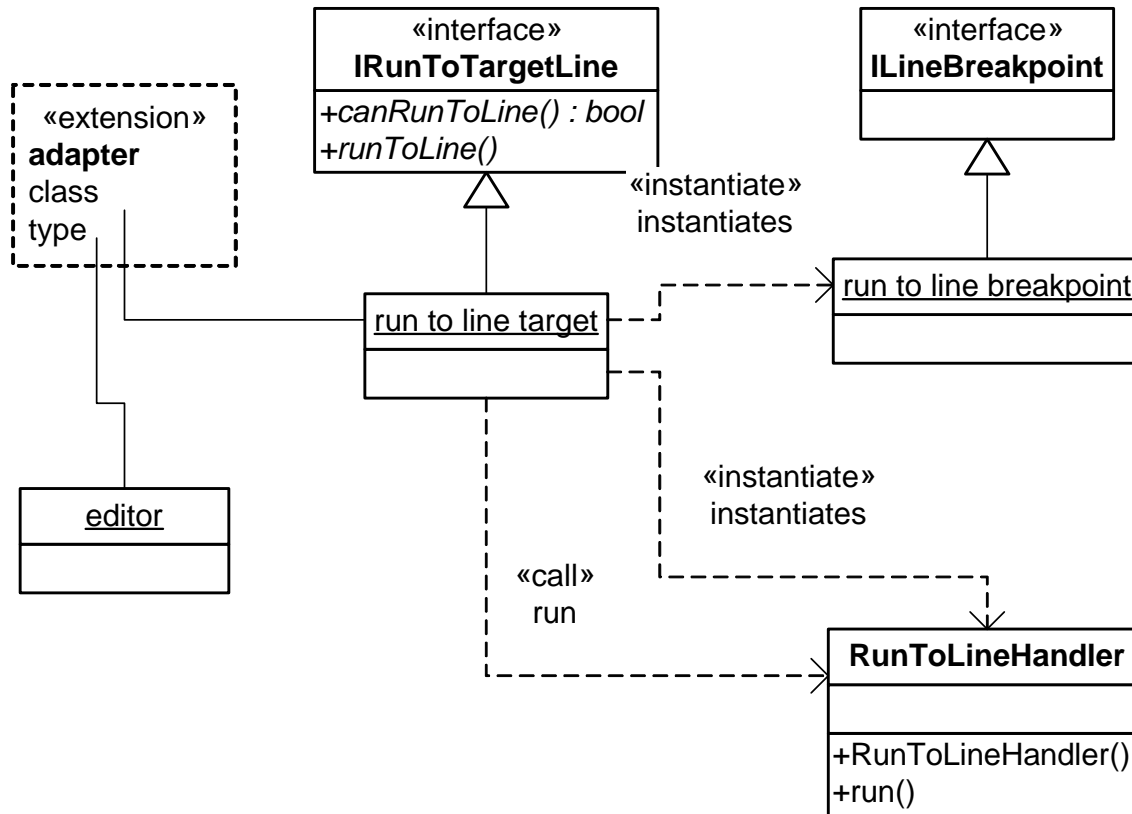
Introduction

- The debug framework provides basic support for user interaction through the editors in the workbench.
 - Run-to-line action
 - Hover for variable values
 - Hover for breakpoints
 - Source line highlighting
 - Double-click in the vertical ruler toggles a breakpoint (see module 3)

1. Run to Line

- The “Run to Line” action in the top level Run menu is retargetable
 - Operates on a `IRunToLineTarget` adapter provided by the active part
 - The platform also provides an action that can be contributed to an editor or view that will operate on that part’s `IRunToLineTarget`
- The platform provides a “run to line” handler for debuggers that implement run to line using line breakpoints (`RunToLineHandler`)
 - Handles the “skip breakpoints during run to line” preference
 - Adds the breakpoint to the target, then resumes the thread, then removes the breakpoint when hit
- Requires a run to line breakpoint
 - A simple subclass of one of your existing breakpoints
 - A breakpoint which is NOT persisted or registered with the manager (and does not show up on the ruler)

1. Run to Line Continued



2. Source Highlighting

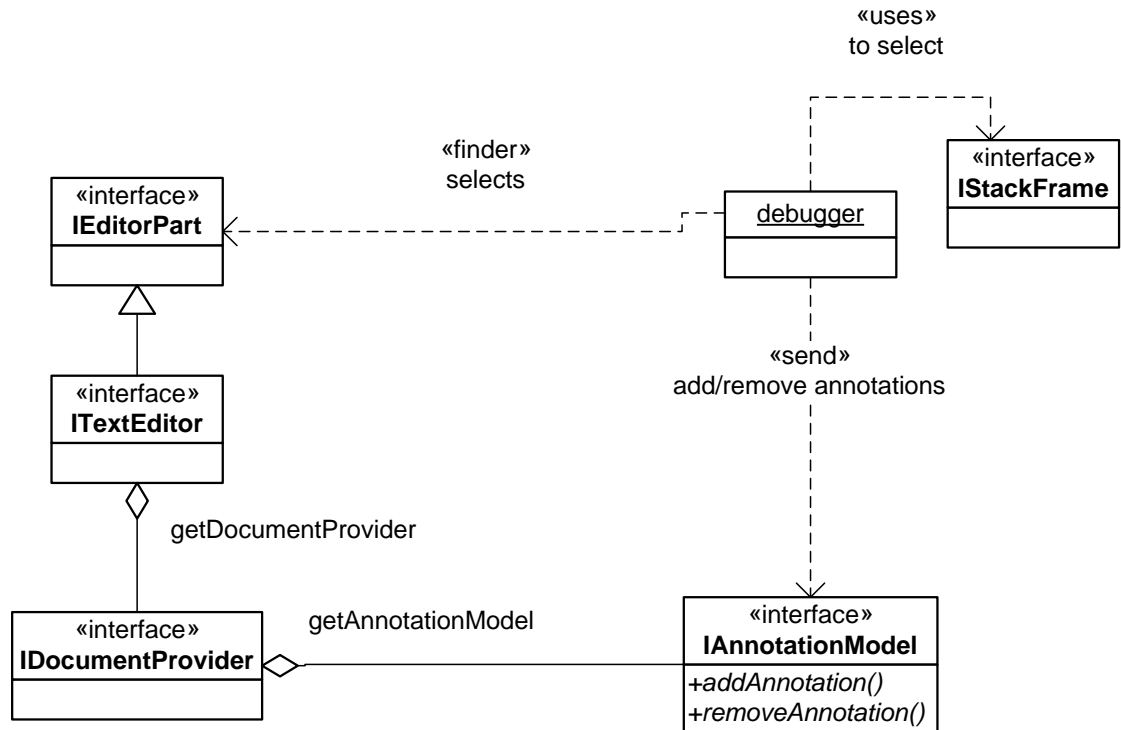
- Can be provided to both text and non-text editors
 - The editor can be an **ITextEditor**, or an editor containing an **ITextEditor**, or a non-text editor
 - Text document provided via the **IDocumentProvider**
- The document provider, an **IDocumentProvider**
 - Provides the document
 - Provides the annotation model
- The annotation model, an **IAnnotationModel**
 - The object that manages all the annotations including source code line highlighting, the breakpoint icons, etc.

2. Source Highlighting Continued

- Text editors
 - Positions the editor to the stack frame line number
 - Highlights to statement (**charStart()**,**charEnd()**) or line (-1,-1)
 - Distinguishes top and non-top stack frames
 - Upon thread resume/terminate automatically removes annotations
 - Colors and styles are preference controllable

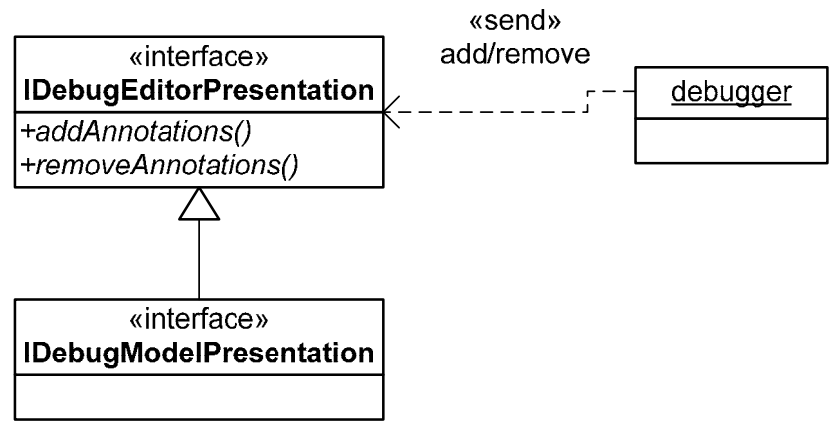
- Non-text Editors
 - The **IDebugEditorPresentation** API provides the escape mechanism
 - Or you implement the Eclipse editor framework with editor, editor input, document provider, and annotation model (describing all of that is out of scope for this tutorial)

2. Source Highlighting Continued



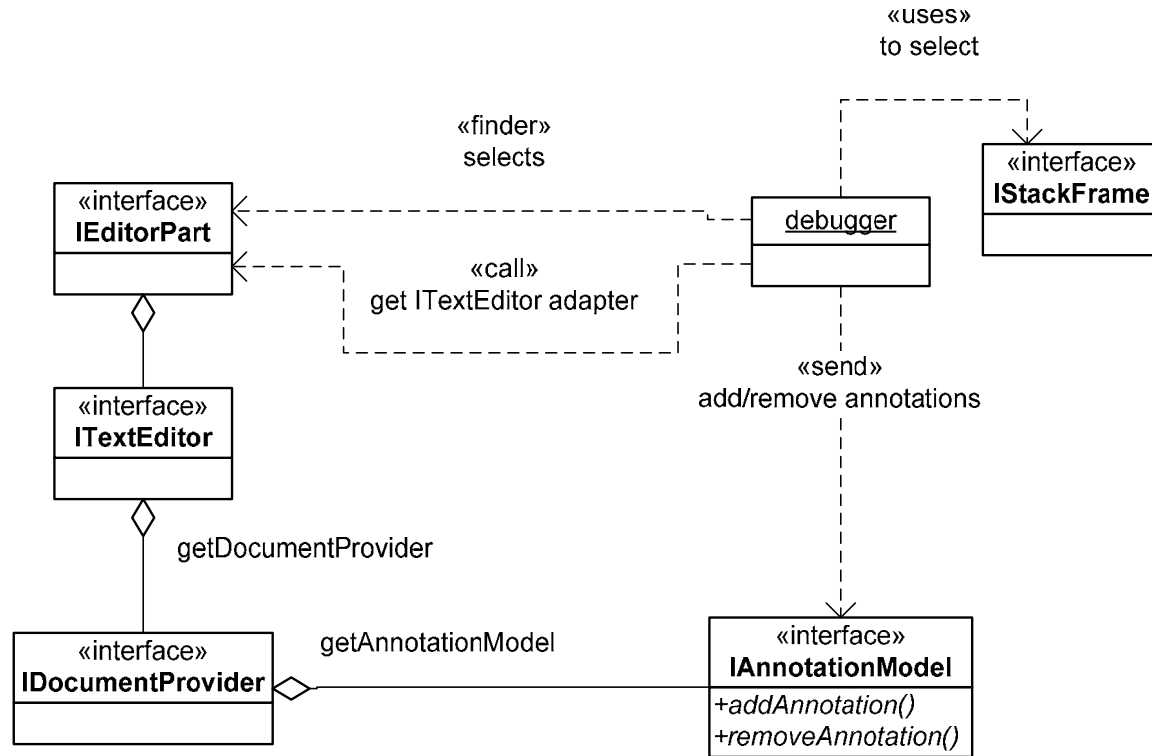
Source Highlighting for Text Editors

2. Source Highlighting Continued



Source Highlighting for Non-Text Editors

2. Source Highlighting Continued



Source Highlighting for Multi-Part Text Editors

New: Instruction Pointer Presentation

- The debug platform provides support to override default instruction pointers displayed in editor rulers.
 - Do this by having your debug model presentation implement the optional interface `IInstructionPointerPresentation`
- Allows instruction pointers to be overridden in one of the following ways
 1. Specify the annotation for an editor and stack frame
 - The annotation is added to the editor at the position (line) indicated by its stack frame
- **public** Annotation
`getInstructionPointerAnnotation(
 IEditorPart, IStackFrame);`

Instruction Pointer Presentation: Continued

2. Specify the type of annotation for an editor and stack frame

- The type corresponds to the identifier of an `<annotationType>` extension.
- The image is controlled by the corresponding `<markerAnnotationSpecification>` extension – which can specify an image in plug-in XML, or specify a delegate class to provide the image

```
public String getInstructionPointerAnnotationType(  
    IEditorPart, IStackFrame);
```

3. Specify the image for an editor and stack frame

```
public Image getInstructionPointerImage(  
    IEditorPart, IStackFrame);
```

Instruction Pointer Presentation: Continued

4. Specify the hover text for an editor and stack frame
 - This can only be done when an annotation type or image was provided
 - When an annotation is provided (1st option), the annotation provides its own text

```
public String getInstructionPointerText(  
    IEditorPart, IStackFrame);
```

3. Hover Info

- The editor
 - Provides the **SourceViewConfiguration**
- The **SourceViewerConfiguration**
 - Provides the **ITextHover** or **ITextHoverExtension**
- The **ITextHover**
 - Determines if the text under the mouse corresponds to anything interesting and if there is an active debug context and if the “anything interesting” is available in the debug context and if all that is true, then return the string to display
- FYI, the Java editor has an extension point for other plug-ins to install hover providers that all operate simultaneously

3. Hover Info Continued

- The Hover Region

```
String varName = null;
try {
    varName = textViewer.getDocument().
                get(hoverRegion.getOffset(),
                    hoverRegion.getLength());
} catch (BadLocationException e) {
    return null;
}
if (varName.startsWith("$")
    && varName.length() > 1) {
    varName = varName.substring(1);
}
...

```

- Hover for Variable Values

- The Debug Context
- Variable Lookup

3. Hover Info Continued

- Hover for Breakpoints
 - Provides the **SourceViewConfiguration** for the editor
 - The **SourceViewerConfiguration** provides the **IAnnotationHover** or **IAnnotationHoverExtension**
 - The **IAnnotationHover** finds something interesting to display about the line the mouse is hovered over
 - **IAnnotationHoverExtension** can provide a custom control while handling multiple annotations on the same line (e.g., overlapping breakpoint, search result, instruction pointer, etc.)

Checklist: Adding Run To Line Support

1. Create a run-to-line breakpoint class as a subclass of your line breakpoint
2. Create an `IRunToLineTarget` adapter that creates an unregistered (1) on an unedited resource (for example, a project).
3. The `runToLine(...)` method of (2) instantiates (1) and a `RunToLineHandler` and passes (1) to the handler.
4. Add (2) as an adapter for your editor using `plugin.xml` (or code)

Checklist: Adding Hover Support

- Create an `ITextHover` that uses `textViewer.getDocument()`. Use `get(hover.getOffset(), hover.getLength())` to retrieve the text and `DebugUITools.getDebugContext()` to retrieve the stack frame.
- Create a `SourceViewerConfiguration` with a `getTextHover()` that creates an instance of (3)
- Call `setSourceViewerConfiguration()` to (4) when creating your editor
- Create an `IAnnotationHover`
- Add (4) to (3)

Checklist: Adding Instruction Pointers

1. If your editor is an `ITextEditor`, your `IStackFrame` does all the work, otherwise your `IDebugModelPresentation` must implement `IDebugEditorPresentation`
2. If you want to override default instruction pointer images, your debug model presentation must implement `IInstructionPointerPresentation`.

Exercise 7: Run to Line

- Goal:
 - Implement run to line in the PDA debugger using a run to line handler
- We provide:
 - A stubbed out run to line target and run to line breakpoint
- Your mission:
 - Define plug-in XML to contribute the run to line target to the PDA editor
 - Add code to instantiate a run to line breakpoint and use it in a run to line handler

Legal Notices

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

IBM, and all IBM-based trademarks are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.