


# Integrating Custom Debuggers into the Eclipse Platform

Darin Wright, Kevin Barnes, and Samantha Chan  
IBM Rational Software

## Your Infrastructure


- To get the most from this tutorial, you need:
  - a laptop capable of running Eclipse and developing plug-ins
  - Eclipse 3.2 I20060315-1200 
  - Tutorial exercises
  - A basic understanding of the debug platform

# Tutorial Structure

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

# Tutorial Modules

1. Displaying Custom Content

 2. Displaying Columns

3. Updating Custom Content

4. Debug Context Management  and Actions

5. The Memory View

6. Other Enhancements In 3.2

## Background: State of the world before 3.2

- A debug model provides implementations of specific interfaces (`IDebugTarget`, etc), and fires debug events in order for the following features to work:
  - View content & update
  - Action enablement
  - Source Lookup
- The Debug view is the single source of the 'active debug context' and must be open to drive a debugger.
- The Variables view and source lookup is hard wired to the selection in the Debug view, and only responds to `IStackFrame` selections.
- Actions (step, terminate, etc.), are hard wired to the selection in the Debug view, and respond to debug events.

## Requests From the Community

- The debug platform is increasingly being used in areas that require more flexible mechanisms than those required by simple Java™ programs
- The DSDP and PTP expressed the need for these features:
  - Flexible debug element hierarchy
  - Pluggable view update policies
  - Asynchronous cancelable interactions between UI and debug model
  - Flexible view wiring (e.g. input to variables view)
  - Ability to debug multiple sessions simultaneously with multiple view sets
  - Introduction of table trees in debug views
  - Drag and drop between all debug views

## The Goal: Support any debug architecture

- The debug platform should support arbitrary debug architectures and implementations of debuggers.
- One possible implementation of a debugger is the “standard element and event based debug model” that exists today.
  - Hierarchy based on `IDebugTarget / IThread / IStackFrame...`
  - View updates based on events
  - API is synchronous, but the UI should interact with it in an asynchronous (non-blocking) fashion
  - View wiring is based on stack frame selection
  - Source lookup is based on stack frame selection

# The Solution: Adapters

- A debugger provides a layer of adapters to interact with the debug platform's views and actions
  - Content and label adapters for view content
  - Model proxy and deltas for view update
  - Capability adapters for actions
  - Source display adapter
- The debug platform provides adapters for implementations of the standard model
  - Existing models continue to work; new models have a choice
- Additionally the platform provides services for debug context management
  - Ways to access and provide the active debug context that drives a debug session

## Evolving The Platform

- The new APIs developed in support of a flexible debug platform are considered provisional for 3.2
- To ensure the APIs meet the needs of the community, we need time for community to use them and provide feedback
  - The APIs are defined in internal packages
  - The APIs will evolve during the next release with the intent to become stable public APIs

# Module 1: Displaying Custom Content

## Introduction: Displaying custom content

- The debug views are capable of displaying arbitrary models
- Most debug views are trees, so this means the views can display arbitrary element hierarchies
- In fact, the debug views can display pretty much anything
  - The debug views aren't just for debuggers any more
- Adapters provide:
  - The elements to display
  - The columns to be displayed
  - The images and labels

## First a word on Adapters

- The Eclipse runtime provides hooks allowing objects to be dynamically extended to provide different interfaces (or “adapters”)
  - `IAdaptable` - the interface adaptable objects implement

```
IAdaptable a = [some adaptable];  
IFoo x = (IFoo)a.getAdapter(IFoo.class);  
if (x != null)  
    [do IFoo things with x]
```

## Where do Adapters come from?

- An object can implement `IAdaptable` directly, but in order to be extensible via platform extension points, objects must subclass `PlatformObject`
- Adapters are created by adapter factories (`IAdapterFactory`)  

```
Object getAdapter(Object adaptable, Class adapterType)  
Class[] getAdapterList()
```
- You can contribute adapters in two ways:
  - Use code to register an adapter factory for a class
  - Use the `org.eclipse.core.runtime.adapters` extension point to register an adapter factory for a class

# Adapter factories

- Use the extension point when extending behavior of classes defined in required plug-ins.
  - This lets the Eclipse runtime discover your adapters even if your plug-in is not loaded

```
<extension point="org.eclipse.core.runtime.adapters">  
  <factory  
    class="org.eclipse.jdt.debug.ui.actions.RetargettableActionAdapterFactory"  
    adaptableType="org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor">  
    <adapter type="org.eclipse.debug.ui.actions.IRunToLineTarget"/>  
    <adapter type="org.eclipse.debug.ui.actions.IToggleBreakpointsTarget"/>  
  </factory>  
</extension point>
```

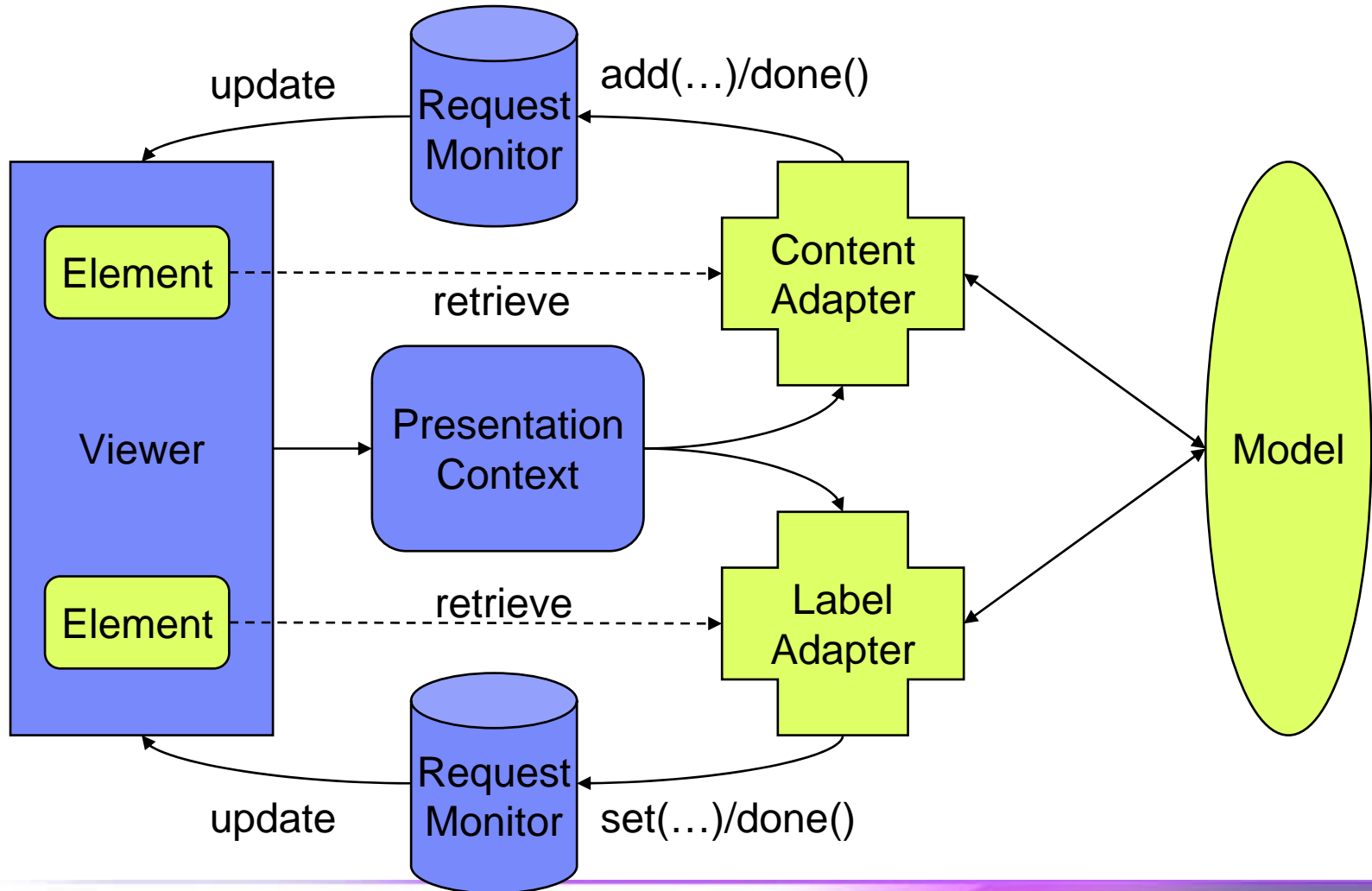
- Use code when defining adapters for classes in your own plug-in

```
IAdapterManager manager = Platform.getAdapterManager();  
manager.registerAdapters(IAdapterFactory, Class)
```

# The Players

- **Viewer**
  - Displays elements from a model
- **Presentation Context**
  - Identifies the view or context in which elements are being displayed
- **Model**
  - Underlying debugger implementation (or any model)
- **Content adapter**
  - Provides children and identifies which elements have children in a given context
- **Label adapter**
  - Provides text and images for model elements in a given context
- **Request monitor**
  - Collects children and labels asynchronously

# The Interactions: Content and Labels



# Checklist: Displaying Custom Content in a View

1. Implement content adapters
  - For each model element implement an `ISynchronousContentAdapter`
2. Implement label adapters
  - For each model element implement an `ISynchronousLabelAdapter`
3. Implement adapter factory
  - Implement an `IAdapterFactory` to instantiate content and label adapters for the model elements
4. Register adapters and adapter factory with the adapter manager
  - Register in code at plug-in startup, or use extension point

## Step 1: Asynchronous Content Adapter

- Asynchronously retrieves children for an element in a specific context, reporting the results to a special progress monitor (request monitor):

```
public interface IAsynchronousContentAdapter {  
  
public void retrieveChildren(  
    Object parent, IPresentationContext context,  
    IChildrenRequestMonitor result);  
  
public void isContainer(  
    Object element, IPresentationContext context,  
    IContainerRequestMonitor result);  
}
```

# Presentation Context

- Describes the context in which a request has been made:  
`IPresentationContext`

```
public IWorkbenchPart getPart( ) ;
```

- The base presentation context describes the part from which the request has been made.
  - Allows adapters to provide different content in different contexts
  - The context is owned/implemented by the view making requests
  - Views may specialize a presentation context to provide more information. For example, the context could describe how many children to retrieve or what columns to retrieve.

# Request Monitor

- Special progress monitor that collects the results of an asynchronous request: `IAynchronousRequestMonitor`

```
public interface IAynchronousRequestMonitor extends  
IProgressMonitor {  
    public void setStatus(IStatus status); }
```

- Each type of request is a specialization of the base request monitor
- Adapters register results with the monitor
- The request can be canceled by the adapter or viewer (need to check `isCanceled()` periodically)
- Must call `done()` when request is complete
- Allows reporting of errors via `setStatus(IStatus)`

## Example: Stack frame content adapter

- Stack frames have different children in different views
  - Debug view = no children
  - Variables view = variables
  - Registers view = register groups

```
public interface IChildrenRequestMonitor extends  
    IAsynchronousRequestMonitor {  
    public void addChild(Object child);  
    public void addChildren(Object[] children);  
}
```

- Child monitor supports incremental content retrieval

# Subclassing AsynchronousContentAdapter

- Implementations must provide content asynchronously
- Platform provides an abstract base class that can be subclassed: `AsynchronousContentAdapter`
  - Provides infrastructure for retrieving content and determining whether an element has children asynchronously for models with synchronous APIs
  - Schedules jobs to do the work

```
protected abstract Object[] getChildren(Object  
parent, IPresentationContext context)
```

```
protected abstract boolean hasChildren(Object  
element, IPresentationContext context)
```

```
protected abstract boolean supportsPartId(String  
id)
```

# Implementation: Stack frame content adapter

```

protected Object[] getChildren(Object parent, IPresentationContext context) throws CoreException
{
    String id = context.getPart().getSite().getId();
    IStackFrame frame = (IStackFrame) parent;
    if (id.equals(IDebugUIConstants.ID_VARIABLE_VIEW)) {
        return frame.getVariables();
    } else if (id.equals(IDebugUIConstants.ID_REGISTER_VIEW)) {
        return frame.getRegisterGroups();
    }
    return EMPTY;
}

protected boolean hasChildren(Object element, IPresentationContext context) throws CoreException
{
    String id = context.getPart().getSite().getId();
    IStackFrame frame = (IStackFrame) element;
    if (id.equals(IDebugUIConstants.ID_VARIABLE_VIEW)) {
        return frame.hasVariables();
    } else if (id.equals(IDebugUIConstants.ID_REGISTER_VIEW)) {
        return frame.hasRegisterGroups();
    }
    return false;
}

protected boolean supportsPartId(String id)
{
    return id.equals(IDebugUIConstants.ID_VARIABLE_VIEW) ||
        id.equals(IDebugUIConstants.ID_REGISTER_VIEW);
}

```

## Step 2: Asynchronous Label Adapter

- Asynchronously computes labels for an element in a specific context, reporting results to a label progress monitor

```
public interface IAsynchronousLabelAdapter {  
  
public void retrieveLabel(  
    Object object, IPresentationContext context,  
    ILabelRequestMonitor result);  
}
```

## Label Request Monitor

- The label request monitor collects text, fonts, image descriptors, and foreground/background colors for each column

```
public interface ILabelRequestMonitor extends
    IAsynchronousRequestMonitor {
public void setLabels(String[] text);
public void setFontDatas(FontData[] fontData);
public void setImageDescriptors(ImageDescriptor[]
    image);
public void setForegrounds( RGB[] foreground);
public void setBackgrounds( RGB[] background);
}
```

# Subclassing AsynchronousLabelAdapter

- Platform provides an abstract base class that can be subclassed: `AsynchronousLabelAdapter`
  - Provides infrastructure for retrieving labels asynchronously for models with synchronous APIs, by scheduling jobs
  - Subclasses can specify whether to use UI jobs (**default is not**)

```
protected abstract String[] getLabels(Object element,  
    IPresentationContext context)
```

```
protected abstract ImageDescriptor[]  
    getImageDescriptors(Object element, IPresentationContext  
    context)
```

```
protected abstract FontData[] getFontDatas(Object element,  
    IPresentationContext context)
```

```
protected abstract RGB[] getForegrounds(Object element,  
    IPresentationContext context)
```

```
protected abstract RGB[] getBackgrounds(Object element,  
    IPresentationContext context)
```

```
protected boolean requiresUIJob()
```

## Compatibility with `IDebugModelPresentation`

- The debug platform registers label adapters for all standard debug model elements
  - Delegates to the appropriate model presentation for each object to compute labels for backwards compatibility
- Clients may continue using `IDebugModelPresentations`
  - May choose to separate model presentation into new adapters
  - Model presentations tend to be large and non-OO in structure, so there may be a benefit in migrating to adapters in terms of code structure
  - The Java debugger uses a mix
    - Original model presentation for its standard elements
    - Adapters for new elements – locks, monitors, thread groups

## Step 3: Adapter Factory

- Implement an adapter factory to instantiate adapters
  - The platform creates only one instance of each adapter to reduce garbage collection.

```
public class DebugElementAdapterFactory implements IAdapterFactory {
    private static IAsynchronousContentAdapter fgAsyncFrame = new
        StackFrameContentAdapter();

    public Object getAdapter(Object adaptableObject, Class adapterType) {
        ...
        if (adapterType.equals(IAsynchronousContentAdapter.class)) {
            ...
            if (adaptableObject instanceof IStackFrame) {
                return fgAsyncFrame;
            }
            ...
        }
    }
}
```

## Step 4: Register Adapter Factory

- Register adapter factory with the platform's adapter manager
  - Usually done in plug-in startup code, unless extending a class from a required plug-in

```
IAdapterManager manager= Platform.getAdapterManager();
DebugElementAdapterFactory propertiesFactory = new
    DebugElementAdapterFactory();
manager.registerAdapters(propertiesFactory, ILaunchManager.class);
manager.registerAdapters(propertiesFactory, ILaunch.class);
manager.registerAdapters(propertiesFactory, IDebugTarget.class);
manager.registerAdapters(propertiesFactory, IProcess.class);
manager.registerAdapters(propertiesFactory, IThread.class);
manager.registerAdapters(propertiesFactory, IStackFrame.class);
```

## Examples: Locks, Monitors & Thread Groups

- The Java debugger contributes custom content and label adapters to display thread groups and lock and monitor information in the debug view with threads
- These customizations are optional user preferences, so content adapters check the presentation context as well as user preferences to determine what content should be displayed

## Chickens and Eggs

- The input to the Debug view is the `ILaunchManager`, which displays all registered launches as root elements
  - To customize content for a launch object, you must provide your own implementation of an `ILaunch` and a content adapter for it
    - The platform registers content/label adapters for `ILaunch`, so you register adapters for your implementation class
  - Customizing content for elements within a launch is simpler, as you only need to register custom adapters for your objects

## Exercise 1: File System Browser

- To demonstrate the flexible content and label features of the debug platform, we have developed a “file system browser” example
  - Your mission: Implement and contribute a content adapter for an `IProject` to display its folders in the debug view

# Module 2: Displaying Columns

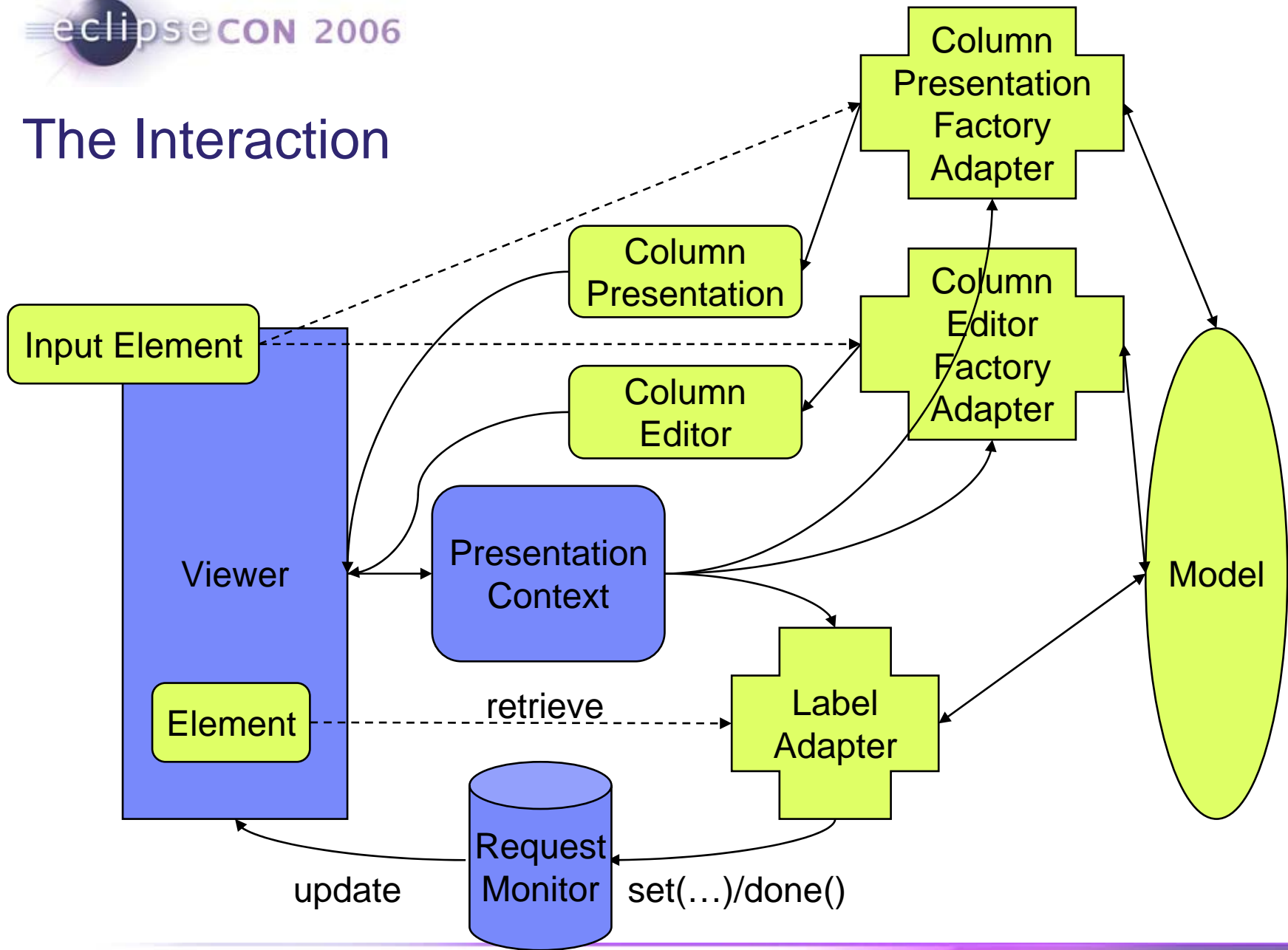
## Introduction: Displaying columns

- Some debug views are capable of displaying arbitrary columns
  - Columns are only supported in homogeneous views (i.e. views that display elements from one model at a time)
    - Variables view
    - Registers view
- Adapters provide:
  - Columns to display
  - Cell editors
  - Labels for each column

# The Players

- **Viewer & Model**
  - (Same as before) Viewer displays elements from a model
- **Presentation context**
  - Identifies the context in which a model is being displayed as well as the columns being displayed
- **Column Presentation**
  - Defines columns, headers and images
- **Column Presentation Factory Adapter**
  - Creates columns presentations for specific contexts
- **Column Editor**
  - Creates cell editors and updates model
- **Column Editor Factory Adapter**
  - Creates column editors for specific contexts
- **Label Adapter**
  - Provides labels for each column in a row

# The Interaction



# Checklist: Displaying Columns

1. Implement column presentation
  - Implement `IColumnPresentation` for each context that you want columns displayed in
2. Register column presentation factory adapter
  - Implement and register `IColumnPresentationFactoryAdapter` to create your column presentations on your debug elements
3. Implement column editor
  - Implement `IColumnEditor` for each context that you want to be able to edit columns in
4. Register column editor factory adapter
  - Implement and register `IColumnEditorFactoryAdapter` to create your column editors on your debug elements
5. Update label adapter to consider columns
  - Update your label adapter to provide labels for each column being displayed

# Step 1: Implement Column Presentation

- A column presentation defines the columns to be displayed
  - Column headers and images – externalized labels
  - Column identifiers – each column is assigned a unique id (`String`)
  - All available columns and initially visible columns
    - Allows many columns to be provided and an initial or default set of columns that should be displayed
    - The user can select which columns to be displayed in the viewer
  - Each column presentation also has a unique id (`String`) to represent its type/kind
    - The column presentation in a viewer is updated when the viewer input changes
    - The column presentation is only changed when the type of columns presentation changes

## The Interface: IColumnPresentation

```
public interface IColumnPresentation {  
    public void init(IPresentationContext context);  
    public void dispose();  
    public String[] getAvailableColumns();  
    public String[] getInitialColumns();  
    public String getHeader(String id);  
    public ImageDescriptor getImageDescriptor(String id);  
    public String getId();  
}
```

## Step 2: Register Column Presentation Factory

- A column presentation factory is an adapter that creates column presentations for specific contexts and elements
  - For example, the platform provides a factory to create a column presentation for `IStackFrame`'s in the Variables view with columns for variable name, declared type, value, and actual type
  - Implement `IColumnPresentationFactoryAdapter`

```
public IColumnPresentation  
    createColumnPresentation(IPresentationContext, Object);  
  
public String  
    getColumnPresentationId(IPresentationContext, Object);
```

- Register the factory with an adapter factory for your debug element

## Step 3: Implement Column Editor

- A column editor allows users to edit cell values in-line
  - Provides custom *cell editors* for each cell
    - The widgetry to modify a value
    - Based on JFace `CellEditor`, which provides standard editors for text, check boxes, combo boxes, etc.
  - Provides a *cell modifier*
    - Determines which cells can be modified, and updates the underlying model after a cell has been edited
    - Based on JFace `ICellModifier`
  - Provides a unique id to represent its *type/kind*
    - The cell editor is only changed when the type of cell editor changes for the viewer input element

## The Interface: IColumnEditor

```
public interface IColumnEditor {  
    public void init(IPresentationContext context);  
    public void dispose();  
    public CellEditor getCellEditor(String id,  
    Object element, Composite parent);  
    public ICellModifier getCellModifier();  
    public String getId();  
}
```

- Provide an implementation of `ICellModifier` to specify which cells can be edited and to update your model
- Provide custom implementations of `CellEditor` where necessary, or re-use implementations provided by `JFace`

## Step 4: Register Column Editor Factory

- A column editor factory is an adapter that creates column editors for specific contexts and elements
  - For example, the platform provides a factory to create a column editor for `IStackFrame`'s in the Variables view
    - The editor's cell modifier allows the value column to be edited when its variable supports value modification via `IValueModification`
    - Uses a simple text editor
  - Implement `IColumnEditorFactoryAdapter`

```
public IColumnEditor createColumnEditor(IPresentationContext,  
    Object);
```

```
public String getColumnEditorId(IPresentationContext, Object);
```

- Register the factory with an adapter factory for your debug element

## Step 5: Update Label Adapter for Columns

- The label adapter provides labels for each column
  - The displayed columns are provided by the presentation context:

```
public String[] getColumns();
```

    - Columns are represented by their identifiers
  - Labels should be returned in the same order as the columns on the label request monitor

```
public void setLabels(String[] text);
```
- Each column also supports foreground/background colors and fonts

# Java Debugger Columns

- The debug platform provides standard columns for the Variables view
  - The Java debugger specializes the support to only allow primitive and string values to be edited in-line
  - The Java debugger provides a drop down editor for `boolean` values using the `ComboBoxCellEditor` provided by `JFace`

## Exercise 2: Columns

- Provide columns for the files in the Variables view
  - Columns: Name, Size, Timestamp, Read-only

# Module 3: Updating Custom Content

## Introduction: Updating Custom Content

- The debug views can display any content
- Most debug models are dynamic in nature
  - A model changes state often without user interaction
  - Examples: thread creation, suspension and termination
- Debug views needs to show an accurate representation of the model as it changes state
  - The model needs a way to inform its view when and how things have changed

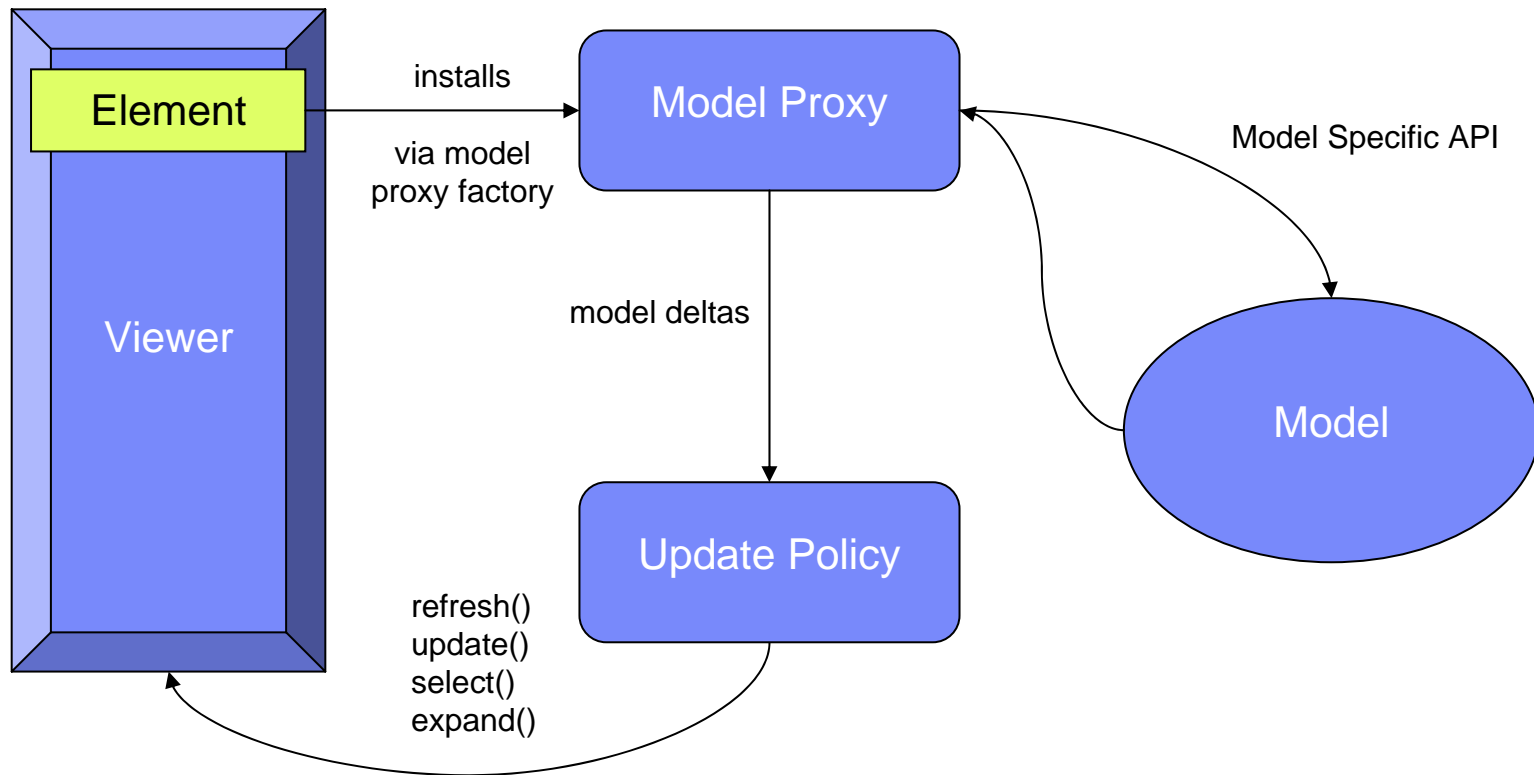
# The Players

- **Viewer**
  - Displays elements from a model
- **Model**
  - Underlying debugger implementation (or any model)
- **Presentation context**
  - Identifies the view or context in which a model is being displayed
- **Model proxy**
  - Interfaces a model with a specific presentation context generating deltas as its model changes
- **Model deltas**
  - Describe how a model has changed
- **Model proxy factory adapter**
  - Creates a model proxies for a model in specific presentation contexts

## The Players (continued)

- **Update policy**
  - Listens to deltas and updates viewer
- **Model selection policy**
  - Resolves selection conflicts within a model
- **Model selection policy factory adapter**
  - Creates model selection policies for specific contexts

# The Interaction



# Checklist: Updating Custom Content

1. Implement model proxy
  - Implement an `IModelProxy` for your debugger in each context that it will be displayed
2. Implement model proxy factory adapter
  - Implement `IModelProxyFactoryAdapter` to instantiate your model proxies for each presentation context it supports
3. Implement model selection policy adapter
  - Implement `IModelSelectionPolicyAdapter` to resolve selection conflicts for your model
4. Implement an adapter factory
  - Implement an `IAdapterFactory` to instantiate your model proxy factory and model selection policy
5. Register adapter factory adapter with the platform's adapter manager

## Step 1: Implement Model Proxy

- A model proxy interfaces a debugger (underlying model) with a specific view: `IModelProxy`
  - Fires deltas as the model changes, specific to a view
  - A model proxy is tied to a presentation context, representing the content your model has provided via its content adapters
  - A model proxy usually represents all elements for a single instance of a model (for example, it represents a single Java debug session and fires deltas for all elements within that mode – threads, frames, etc.)
    - You could choose to implement model proxies for each element in a model

## The Interface: IModelProxy

```
public interface IModelProxy {  
  
    public void init(IPresentationContext context);  
    public void installed();  
    public void dispose();  
    public void  
        addModelChangeListener(IModelChangeListener  
            listener);  
    public void  
        removeModelChangeListener(IModelChangeListener  
            listener);  
  
}
```

## Subclassing AbstractModelProxy

- The debug platform provides an abstract implementation of `IModelProxy` that should be subclassed: `AbstractModelProxy`. It provides support for:
  - Adding and removing model delta listeners
  - Firing deltas to all registered listeners
  - Access to the context in which it has been installed
- The debug platform provides implementations of model proxies for the standard debug model and standard views
  - For example, a model proxy is provided for `IDebugTarget`'s in the debug view. The proxy listens to debug events from a specific debug session and fires corresponding deltas.

# Model Deltas

- An `IModelDelta` describes incremental changes in a model
  - Similar to a resource delta (`IResourceDelta`)
  - Hierarchical description of changes – a tree of nodes rooted at the viewer input element
  - Each node references a model element and describes how the element changed (if at all), and what action (if any) to perform on the element
- Changes and actions are described by flags in each delta node
  - Change flags: `ADDED`, `REMOVED`, `REPLACED`, `INSERTED`, `CONTENT`, `STATE`
  - Action flags: `SELECT`, `EXPAND`
- Platform provides an implementation of model deltas to be instantiated by model proxy implementations: `ModelDelta`

## Example: Launch Manager Model Proxy

- The launch manager model proxy interfaces registered launches with the debug view
  - Listens to launch notifications (`ILaunchesListener2`), and creates corresponding deltas

```
public void launchesAdded(ILaunch[] launches) {  
    fireDelta(launches, IModelDelta.ADDED | IModelDelta.EXPAND);  
}  
public void launchesChanged(ILaunch[] launches) {  
    fireDelta(launches, IModelDelta.STATE | IModelDelta.CONTENT);  
}  
public void launchesTerminated(ILaunch[] launches) {  
    fireDelta(launches, IModelDelta.CONTENT);  
}  
public void launchesRemoved(ILaunch[] launches) {  
    fireDelta(launches, IModelDelta.REMOVED);  
}
```

## Example cont'd

- Creating and firing model deltas

```
protected void fireDelta(ILaunch[] launches, int launchFlags) {
    ModelDelta delta = new ModelDelta(fLauncherManager,
        IModelDelta.NO_CHANGE);
    for (int i = 0; i < launches.length; i++) {
        delta.addNode(launches[i], launchFlags);
    }
    fireModelChanged(delta); // inherited from AbstractModelDelta
}
```

- When installed all existing launch objects are expanded

```
public void installed() {
    ILaunch[] launches = fLauncherManager.getLaunches();
    if (launches.length > 0) {
        fireDelta(launches, IModelDelta.EXPAND);
    }
}
```

## Step 2: Implement model proxy factory adapter

- The `IModelProxyFactoryAdapter` instantiates model proxies for elements in specific contexts
  - (Note: the model proxy itself could not be an adapter because we need to create context specific proxies, and the `getAdapter(...)` infrastructure does not allow for creating different adapters based on context)

```
public interface IModelProxyFactoryAdapter {  
public IModelProxy createModelProxy(  
    Object element, IPresentationContext context);  
}
```

# Example model proxy factory adapter

```
public class DefaultModelProxyFactory implements IModelProxyFactoryAdapter {  
  
    public IModelProxy createModelProxy(Object element, IPresentationContext context) {  
        IWorkbenchPart part = context.getPart();  
        if (part != null) {  
            String id = part.getSite().getId();  
            if (IDebugUIConstants.ID_DEBUG_VIEW.equals(id)) {  
                if (element instanceof IDebugTarget) {  
                    return new DebugTargetProxy((IDebugTarget)element);  
                }  
                if (element instanceof ILaunchManager) {  
                    return new LaunchManagerProxy();  
                }  
                ...  
            }  
        }  
    }  
}
```

## Step 3: Implement model selection policy

- The `IModelSelectionPolicy` resolves selection conflicts for a model
  - When a model delta instructs a viewer to set the selection, and there is an existing selection, the viewer consults the selection policy of the existing selection
  - The new selection is set only if:
    - The new selection is contained in the same model as the existing selection and the new selection should override the existing selection
    - Or the new selection is in a different model and the existing selection is not “sticky”

## The Interface: IModelSelectionPolicy

```
public boolean contains(ISelection selection,  
    IPresentationContext context);
```

```
public boolean overrides(ISelection existing,  
    ISelection candidate, IPresentationContext  
    context);
```

```
public boolean isSticky(ISelection selection,  
    IPresentationContext context);
```

## Example Selection Policy

- The debug platform provides a default selection policy for debug elements in the debug view
  - The policy ensures that when a thread suspends, the newly suspended thread is only selected if there is not a currently suspended stack frame selected
    - The user's focus on a suspended context is not changed as other contexts suspend

## Selection Policy Factory

- A selection policy is created from a selection policy factory:  
IModelSelectionPolicyFactoryAdapter
- Allows context specific selection policies to be created (i.e. for each view and model)

```
public IModelSelectionPolicy  
createModelSelectionPolicyAdapter(  
    Object element,  
    IPresentationContext context);
```

## Steps 4 & 5: Create & Register Adapter Factory

- This is the same as in module 1
  - Create an implementation of `IAdapterFactory` to instantiate your `IModelProxyFactoryAdapter` and `IModelSelectionPolicyAdapter` implementations for your model
  - Register your adapter factory in code or using an extension

# The Java Debugger Model Proxy

- The Java debugger provides its own model proxy to accommodate thread groups
  - The thread groups introduce non-standard elements into the debug element hierarchy and must be accounted for when creating element deltas (i.e. paths to threads and stack frames)
  - Current the Java debugger just subclasses the platform's `DebugTargetProxy` and overrides a couple methods to build the correct paths to its model elements

## Exercise 3: Build a Model Proxy

- Build a model proxy for the “file system browser” example that updates the presentation as the file system changes:
  - Translate `IResourceDelta`'s into `IModelDelta`'s for the Debug view

# Module 4: Debug Context Management and Actions

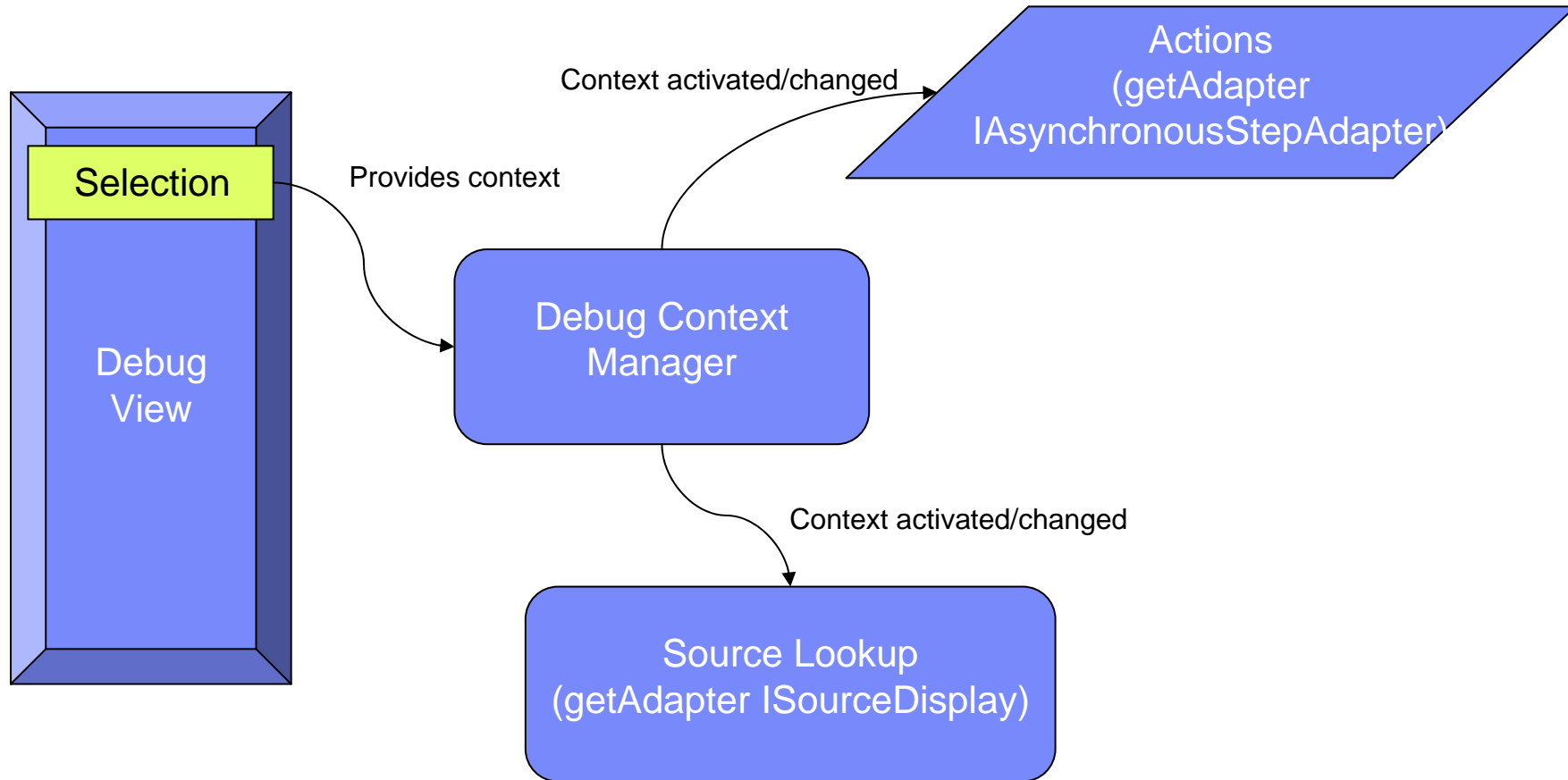
# Introduction: Debug Context Management

- A debug context represents an execution context
  - For example, the top stack frame in a suspended thread, an executing thread, or an entire debug target
- The debug platform provides facilities for driving and responding to the active context
  - The active context drives the debug user interface
  - Provides the context for step actions, etc., variables displayed, and source lookup

# The Players

- **Context manager**
  - There is one debug context manager
  - Manages the active context in each window, providing change notification to registered listeners
- **Context provider**
  - Provides an active context for a workbench part
  - Registers with the debug context manager, provides notification as its active context change
- **Context listener**
  - Notified of changes in the active context
  - Registers with the debug context manager for change notification in a window or a specific part

# The Interaction



## Example Context Provider: The Debug View

- A context provider
  - Notifies when a context is activated
  - When the active context has changed state
- The Debug view is a context provider
  - Registers itself with the context manager
  - The selection in the debug view provides its active context
  - As the selection changes it notifies its listener (context manager)
- Driving the debug view
  - User actions change the context – i.e. explicit selection
  - Model proxies drive the debug view by firing deltas with the `IModelDelta.SELECTION` and `IModelDelta.STATE` flags

## Example Context Listener: Step Action

- The Step action in the top-level “Run” menu
  - Registers as a context listener with its workbench window
  - When notified of a context activation or state change, it updates its enablement by retrieving the “`IASynchronousStepAdapter`” from the context (if any)
  - The step action in the Debug view is similar, but registers as a context listener in a specific part (its view)

## Determining The Active Context

- There can be many context providers in a window
  - There can be an active context in each provider (part)
- There is only ***one active context per window***, based on the active part
  - If the active part is a context provider, it drives the context in its window
  - If the active part is not a context provider, the last active part that was a context provider drives the active context in that window

# The Interface: IDebugContextManager

- Registering providers and listeners (have corresponding remove methods)

```
public void addDebugContextProvider(IDebugContextProvider provider);  
public void addDebugContextListener(IDebugContextListener listener,  
    IWorkbenchWindow window);  
public void addDebugContextListener(IDebugContextListener listener,  
    IWorkbenchWindow window, String partId);
```

- Accessing the active context

```
public ISelection getActiveContext(IWorkbenchWindow window);  
public ISelection getActiveContext(IWorkbenchWindow window, String partId);
```

- Similar to selection service in the workbench
  - The context is represented as a selection
  - Allows for “multi-contexts” (i.e. multi-selection)

# Everything revolves around the active context

- The active context drives debugging
  - The actions listen and act on the active context
  - Source lookup listens to and displays the active context
  - The Variables view is populated by the active context
  - The Expression view computes watch expression values based on the active context
  - Debug view management (auto open/close) is based on the active context

## The Net Effect

- Actions listening to the active context are retargettable
  - Actions used to listen to the Debug view selection explicitly
  - Now they listen to the active context and work on adapters, which could be provided by anyone
- View wiring is flexible
  - Variables view used to listen to stack frame selection in Debug view explicitly, now the active context is the input to the view and can provide content (based on content adapters)
- Source lookup is pluggable
  - Source lookup used to be hardwired to the Debug view, but is now driven by the active context's `ISourceDisplayAdapter`

# Checklist: Integration with Debug Contexts

## 1. Provide standard debug capability adapters

- Implement standard debug capability interfaces as in 3.1
  - `IStep`, `ITerminate`, `ISuspendResume`, `IDisconnect`, `IDropToFrame`
- Or provide and register implementations of new asynchronous adapters:
  - `IASynchronousStepAdapter`,  
`IASynchronousTerminateAdapter`, etc.,

## 2. Provide content adapters to display non-standard content

- For example, to display variables when a thread is selected, provide an `IASynchronousContentAdapter` for your thread in the Variables view presentation context

## Checklist: cont'd

3. Provide `ISourceDisplayAdpater` for non-standard source lookup
  - The platform provides an adapter for `IStackFrame` that displays source when a frame is activated/changed, and clears instruction pointers when its thread resumes/terminates.
4. Register any context providers with context manager
  - If you have any special views that need to drive the active debug context, they need to register with the debug context manager and implement `IDebugContextProvider`.
  - For example, CDT plans to implement non-standard stepping in a disassembly view
    - The view will be a context provider
    - The context it provides will have an `IStep` adapter that steps over assembly instructions rather than lines

## One More Integration Point

- Debug models are no longer required to fire debug events
  - When your model implements the standard debug capabilities (`IStep`, etc.), you no longer have to fire events
- How does the platform know when your debugger has suspended?
  - The debug view works based on model deltas, but that does not tell the debug platform when to switch perspectives.
  - `ILaunch` implementations must provide an `ISuspendTriggerAdapter` if they do not contain standard debug models.
  - The platform registers itself as an `ISuspendTriggerListener` with each registered launch's suspend trigger adapter, so it knows when to change perspectives (or prompt, etc).

# Exercises

1. Implement the suspend/resume adapter for folders
  - Suspend makes the folder read-only
  - Resume makes the folder read-write

# Module 5: The Memory View

## Introduction

- Memory View shows content of a memory monitor
- Goal: A flexible view to allow clients to render content of a memory monitor in any way that is suitable for their purposes.
  - E.g. Table Viewer, Tree Viewer, Bitmap, etc
- Solution: Memory View and a framework to allow clients to contribute different ways of looking at a memory monitor

# Who hasn't seen the Memory View?

The screenshot displays the Eclipse Memory View window, which is split into two panes. The left pane shows the memory address space hierarchy, and the right pane shows the memory data in hex and ASCII formats.

**Memory View - Hex View (a : 0x6880160D <Hex>)**

Address	0 - 3	4 - 7	8 - B	C - F
68801600	3BE9C515	EB1B9F34	C7D7584A	EB744FD7
68801610	23E1035E	6FAAEA3C	3C73FA35	35C79748
68801620	DE3F7DA5	A0A8E618	91426219	B074C0FF
68801630	EDAAF7CB	74385F00	7C561B47	7156E50C
68801640	A1C79CC3	086241DF	17B7BB1B	2CA1E327
68801650	1DB9D5EC	344B095B	118773BF	47372FDC
68801660	B4EB8D55	EAA552BC	1D67FD54	A7398833
68801670	140CFCFC	4C68D88B	7F197379	B1E17DDD
68801680	CA5BE4A0	7BC13E44	2066F5DA	F38F1A5E
68801690	CB245F9A	A64B6A49	9442D0B7	0121359E
688016A0	1F8960EF	1C7C5393	8D2DD182	418763C3
688016B0	0C855687	E761CC6E	71DB7D06	F095FA59
688016C0	4F2E0665	8E8D6398	9D16A941	1E1B074F
688016D0	9F39E34A	764EF323	70E3EF6C	D9324CD8

**Memory View - ASCII View (a : 0x6880160D <ASCII>)**

Address	0 - 3	4 - 7	8 - B	C - F
68801600	;éÄ¹	ë+ÿ4	Ç×XJ	ä ètO×
68801610	#áL^	o*è<	<su5	5Ç-H
68801620	È?)¥	"æ†	'Bb†	°tÀÿ
68801630	í*÷Ë	t8_r	V+G	qVâ□
68801640	¡ÇœÄ	▣bA8	†·»+	,iä'
68801650	°Öi	4K[	◀#sz	G7/Û
68801660	'ëOU	ê¥Rw	qýT	§9^3
68801670	¶¶üü	LhØ	□ sy	±á)Ý
68801680	È[ä	{Ä>D	fðÜ	ó[+^
68801690	Ë\$_š	¡KjI	"BØ·	¡!5ž
688016A0	š'i	S"	□-Ñ,	A+cÄ
688016B0	□.V+	çain	qÛ)-	š·úY
688016C0	O.-e	Ž□c"	[T@A	←•O
688016D0	Ý9ãJ	vNó#	pãil	Û2LØ
688016E0	ÝÄ¿o	X"ð□	+äxγ	úØ!-

# The Players

## ■ Core

- Memory Block – a memory monitor from a debug model
- Memory Block Retrieval – for creating a memory block
- Memory Block Manager – keeps track of memory monitors from the workbench

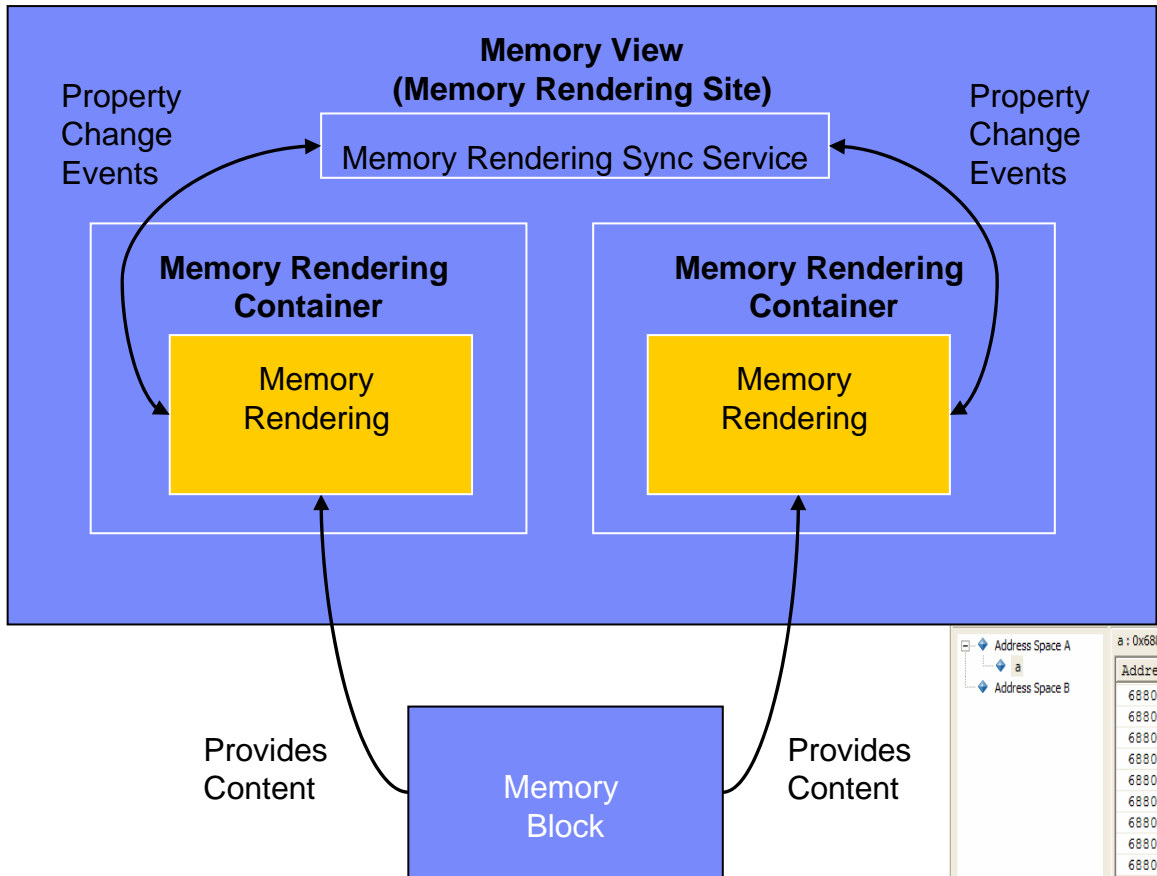
## ■ UI

- Memory Renderings – UI artifact that displays content of a memory block
- Memory Rendering Container – the container within a memory rendering site that hosts a rendering
- Memory Rendering Site – a workbench site that hosts memory renderings, has multiple memory rendering containers
- Memory Rendering Synchronization Service – Allow renderings from the same site to synchronize with each other. (e.g. scrolling, selection, etc.)

## ■ Framework

- Memory Rendering Type
  - Defines the name and other attributes of a type of memory rendering
  - Defines a delegate for creating a memory rendering
- Memory Rendering Manager
  - Keeps track of available memory rendering types
  - Allows for clients to query for a list of valid rendering types for a type of memory block

# The Memory View



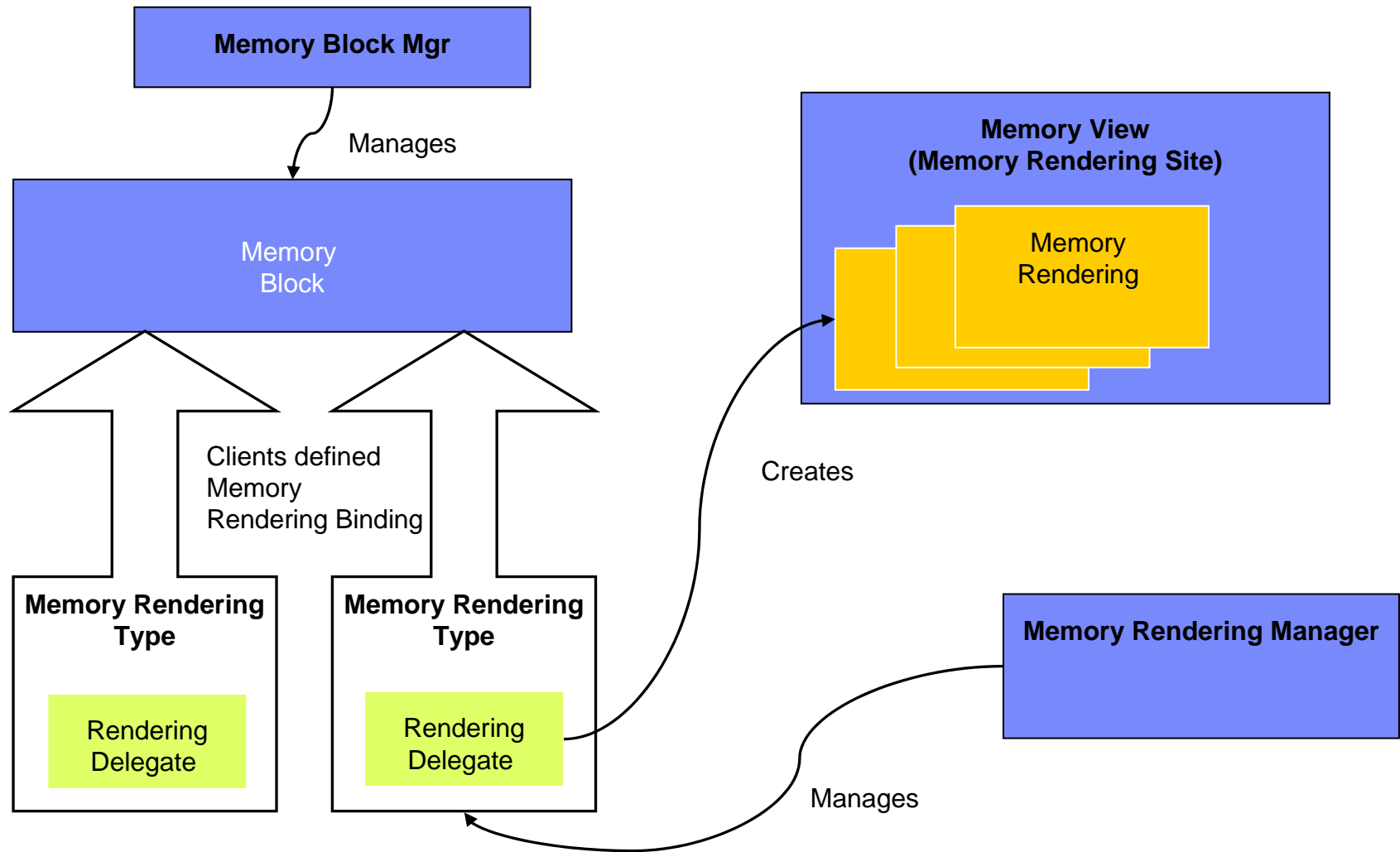
Address Space A

Address	0 - 3	4 - 7	8 - B	C - F
68801600	3BE9C515	EB1B9F34	C7D7584A	EB744ED7
68801610	23E1035E	6FAAE3C	3C73FA35	35C79748
68801620	DE3F7DA5	A0A8E618	91426219	B074C0FF
68801630	EDAAF7CB	74385F00	7C561B47	7156E50C
68801640	A1C79CC3	086241DF	17B7BB1B	2CA1E327
68801650	1DB9D5EC	344B095B	118773BF	47372FDC
68801660	B4EB8D55	EAA552BC	1D67FD54	A7398833
68801670	140CF3FC	4C68D88B	7F197379	B1E17DDD
68801680	CA5BE4A0	7BC13E44	2066F5DA	F38F1A5E
68801690	CB245F9A	A64B6A49	9442D0B7	0121359E
688016A0	1F8960EF	1C7C5393	8D2DD182	418763C3
688016B0	0C855687	E761CC6E	71DB7D06	F095FA59
688016C0	4F2E0665	8E8D6398	9D16A941	1E1B074F
688016D0	9F39E34A	764EF323	70E3EF6C	D9324CD8

Address Space B

Address	0 - 3	4 - 7	8 - B	C - F
68801600	·éÁL	ë+ÿ4	Ç×XJ	·etcX
68801610	·#á^	o*è<	<au\$	5Ç-H
68801620	·?·¥	·"·f	·'Bb	·t·ÿ
68801630	·i·+·È	·t8_·r	· V·+G	·qV·d
68801640	·j·Ç·Ä	·D·b·A	· ··»·+	·,·j·á
68801650	·'ò·i	·4K[	·«·s2	·G7/Ü
68801660	·'e·U	·é·ÿ·R·+	·g·ÿ·T	·\$9·3
68801670	· ·ÿ·ü·ü	·L·h·0	· ·ÿ·s·y	·±·á·ÿ
68801680	·È·[·á	·(·Á·>	·f·ò·Ü	·6·>·^
68801690	·È·\$·_	· ·K·J·I	·'·B·B·	·r·!·5·z
688016A0	·h·'·i	· ·S·"	·Q·-·N·	·A·t·c·Ä
688016B0	·L·V·+	·c·a·i·n	·q·Ü·)	·è·ú·ÿ
688016C0	·O·-·e	·Z·C·"	· ·ÿ·@·A	·+·*·0
688016D0	·'·ÿ·Á·J	·v·N·6·#	·p·á·i·l	·Ü·Z·L·0
688016E0	·'·ÿ·Á·J	·X·"·d·0	·+·á·x·f	·ú·0·J·-

# The Framework



## How to use the Memory View?

- Use renderings provided by the platform
  - Provide a set of renderings: Hex, ASCII, Signed Integer and Unsigned Integer
  - EASY: implement memory block and creates required binding to existing renderings
- Create custom renderings and have them hosted by Memory View
  - HARDER: implement memory block and own rendering types. Also need to create binding.
  - Platform provides abstract implementation to table rendering and text rendering.

# Checklist

- Reusing platform's memory renderings
  1. Implement IMemoryBlockRetrieval
  2. Implement IMemoryBlock
  3. Create memory rendering binding with the memory block
    - Create a binding for each memory rendering to be used
  4. Customize the rendering as needed

## Step 1: Implement IMemoryBlockRetrieval

- Interface to allow UI to ask for a memory block from a model
- Usually implemented in the debug target. Clients may optionally create a IMemoryBlockRetrieval adapter to separate out the memory block retrieval capabilities.

**IMemoryBlockRetrieval:**

```
public boolean supportsStorageRetrieval();  
IMemoryBlock getMemoryBlock(long startAddress, long length) throws DebugException;
```

**IMemoryBlockRetrievalExtension:**

```
public IMemoryBlockExtension getExtendedMemoryBlock(String expression, Object context)  
throws DebugException;
```

# What should I implement?

## **IMemoryBlockRetrieval**

## **vs IMemoryBlockRetrievalExtension**

- Creates IMemoryBlock –a basic static memory block. Does not have all the capabilities, but easier to implement.

- User must enter an address and length to create a memory monitor.

- Content of the resulting memory monitor must be retrieved as a whole, does not allow dynamic scrolling.

- Creates IMemoryBlockExtension – provides much more capabilities and information. Harder to implement.

- User can enter an arbitrary expression when creating a memory monitor (e.g. variable name)

- Resulting memory monitor can retrieve content dynamically and hence enables dynamic scrolling

## Step 2: Implement IMemoryBlock

- Fixed Length
  - Does not allow views to load memory dynamically as #getBytes get the entire content of the memory block
- Limited address size
  - address can be as big as what a long can represent (32-bit addressing)
- Limited to byte-size addressable unit
- Cannot specify endianness

```
public long getStartAddress();  
public long getLength();  
public byte[] getBytes() throws DebugException;  
public void setValue(long offset, byte[] bytes) throws  
DebugException;
```

## Step 2: Implement IMemoryBlockExtension

- Not fixed to 32-bit addressing. Addresses are represented by BigInteger instead.

```
public BigInteger getBigBaseAddress() throws DebugException;  
public int getAddressSize() throws DebugException;
```

- Allow models to specify a boundary on the memory block.

```
public BigInteger getMemoryBlockStartAddress() throws DebugException;  
public BigInteger getMemoryBlockEndAddress() throws DebugException;
```

- Support addressable units bigger than 1 byte. Allows models to specify the size of its smallest addressable unit.

```
public int getAddressableSize() throws DebugException;
```

- Allow clients to ask for memory dynamically as needed. The memory block does not really have a fixed length.

```
public MemoryByte[] getBytesFromAddress(BigInteger address, long units) throws DebugException;  
public MemoryByte[] getBytesFromOffset(BigInteger unitOffset, long addressableUnits) throws  
    DebugException
```

- Allow models to provide more information about their environment.
- MemoryByte allows model to tag each byte with attributes that describe the byte. MemoryByte has a value and its attributes.

## Step 3: Create a memory rendering binding

- Memory Rendering Binding defines what types of memory rendering are applicable for a type of memory block.
- Use the memory rendering extension point to create a memory rendering binding.
- Renderings types provided by the platform:
  - Raw Memory: `org.eclipse.debug.ui.rendering.raw_memory`
  - ASCII: `org.eclipse.debug.ui.rendering.ascii`
  - Signed Integer: `org.eclipse.debug.ui.rendering.signedint`
  - Unsigned Integer: `org.eclipse.debug.ui.rendering.unsignedint`

```
<extension point="org.eclipse.debug.ui.memoryRenderings">
  <renderingBindings
    defaultIds="org.eclipse.debug.ui.rendering.raw_memory,org.eclipse.debug.ui.rendering.ascii"
    primaryId="org.eclipse.debug.ui.rendering.raw_memory"
    renderingIds="org.eclipse.debug.ui.rendering.raw_memory,org.eclipse.debug.ui.rendering.ascii"
    <enablement>
      <instanceof value="com.abc.sampleadapter.extendedMemoryBlock.SampleExtendedMemoryBlock"/>
    </enablement>
  </renderingBindings>
```

## Step 4: Customizing the renderings

- *Text, images*
  - provide an ILabelProvider adapter from your memory block
- *Font*
  - provide an IFontProvider adapter from your memory block
- *Color*
  - provide an IColorProvider adapter from your memory block
- *Column and row headings*
  - provide an IMemoryBlockTablePresentation adapter from your memory block
- Adapter will be asked to render a *MemoryRenderingElement*.
- MemoryRenderingElement describes the content of a cell in a table rendering. It provides the following information:
  - The rendering that is showing this cell.
  - MemoryByte[] to be rendered
  - Address of where these bytes are located.

## Exercise 1

- Implement a memory block
- Reuse all renderings from the platform
- Default to show the HEX and signed integer rendering when a memory block is created
- Customize to show read-only memory in a different color
- Customize the column and row headings in a table rendering

# Checklist

- Creating your own rendering type
  1. Extend `org.eclipse.debug.ui.memoryrenderings` – create own rendering type
  2. Implement `IMemoryRenderingTypeDelegate`
  3. Implement `IMemoryRendering`
  4. Extend `org.eclipse.debug.ui.memoryrenderings` – bind the memory block to the new rendering type

## Step 1 – Create Rendering Type

- Extend *memoryRenderings* extension point and define a *renderingType*.
- A rendering type consists of:
  - Java class that implements `IMemoryRenderingTypeDelegate`
    - `public IMemoryRendering createRendering(String id) throws CoreException;`
  - Name of the rendering
  - Unique identifier of the rendering.
- Example: Hex Rendering

```
<extension point="org.eclipse.debug.ui.memoryRenderings">  
<renderingType  
  class="org.eclipse.debug.internal.ui.views.memory.renderings.HexRenderingTypeDelegate"  
  name="%RawHex"  
  id="org.eclipse.debug.ui.rendering.raw_memory"/>
```

## Step 2 – Implement IMemoryRenderingTypeDelegate

- IMemoryRenderingTypeDelegate is responsible for creating the actual memory rendering.
- Usually called by actions for creating a rendering or by the memory rendering manager.
- You are only responsible to create the memory rendering instance. Caller of this method will be responsible to initialize the rendering when necessary.
  - ```
public IMemoryRendering createRendering(String id)  
    throws CoreException;
```

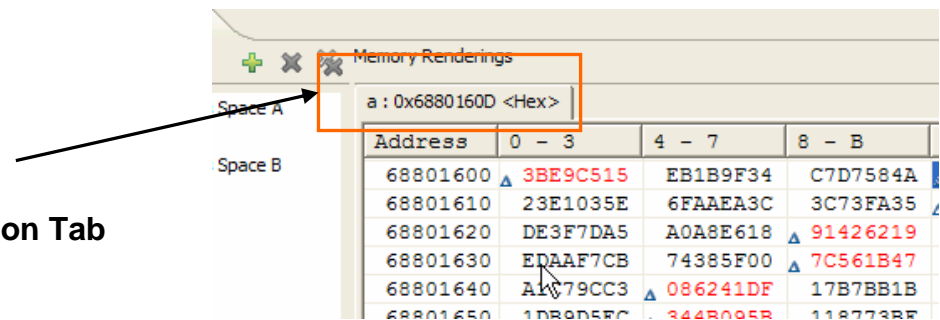
## Step 3 – Implement IMemoryRendering

- IMemoryRendering is responsible for rendering and displaying the content of a memory block.
- Clients may reuse AbstractMemoryRendering class.
- Life Cycle:
  - Initialization:
    - `public void init(IMemoryRenderingContainer container, IMemoryBlock block);`
  - Create Control:
    - `public Control createControl(Composite parent);`
  - Dispose:
    - `public void dispose();`

## Step 3 – Implement IMemoryRendering – Image & Label

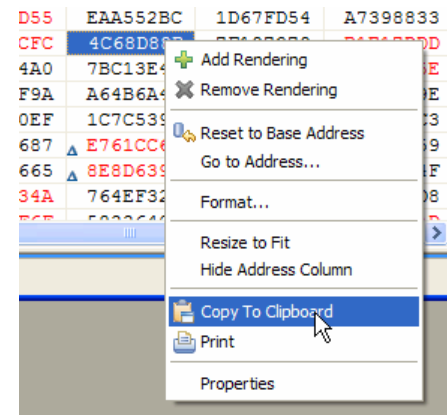
- IMemoryRendering provides a label and an image:
  - `public String getLabel();`
  - `public Image getImage();`
- AbstractMemoryRendering provides a default label.
- For changing the label and image on the tabs dynamically, fire the following `propertyChanged` event:
  - `IBasicPropertyConstants.P_TEXT` – for modifying labels
  - `IBasicPropertyConstants.P_IMAGE` – for modifying images

Label and Image on Tab



## Step 3 – Implement IMemoryRendering – Context Menu

- Clients may contribute a pop-up context menu action via `viewerContribution` or `objectContribution`.
- To ensure that a rendering is capable of showing a context menu, a rendering must create the context menu for its controls.
- `AbstractMemoryRendering` provides a helper method for creating context menu:
  - `protected void createPopupMenu(Control control)`
- For each control that needs to show context menu, call this helper method.
- The context menu id is the memory rendering container's id.



## Step 4 – Create Memory Rendering Binding

- Similar to reusing the renderings provided by the platform, bind your memory block with the new memory rendering type.

```
<extension point="org.eclipse.debug.ui.memoryRenderings">  
  <renderingBindings renderingIds="my.new.rendering.id"/>  
    <enablement>  
      <instanceof value="my.memory.block"/>  
    </enablement>  
  </renderingBindings>
```

## Exercise 2

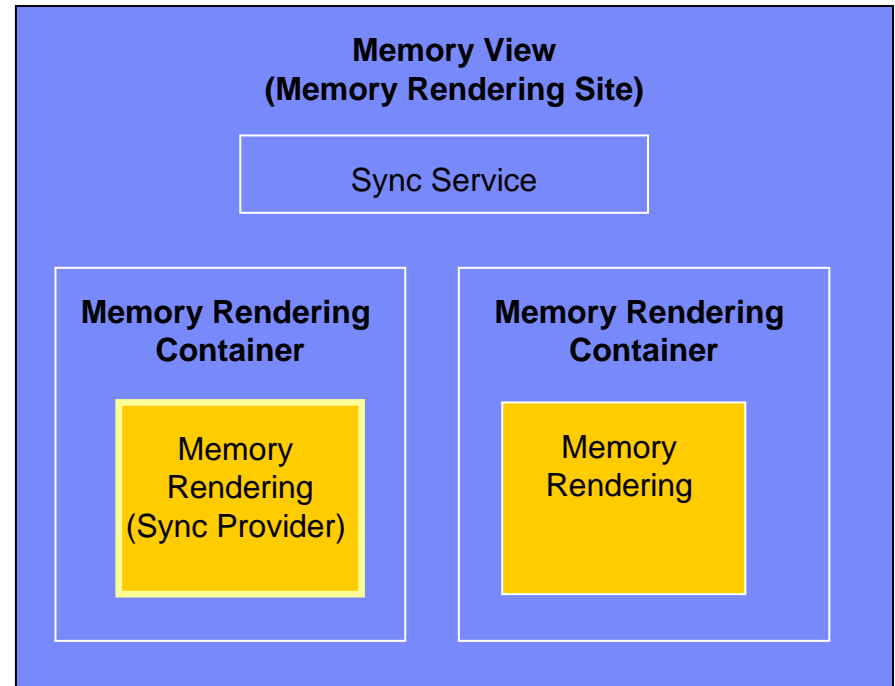
- Create a customized rendering that displays content of a memory block in a tree.
- Optional:
  - Create a context menu for your rendering.
  - Add an image to the tab label.

# Synchronization between renderings

- Synchronization is needed to provide a better user experience.
  - User does not have to manually keep the renderings in sync.
- Challenges:
  - Renderings come and go – hard to keep track of what renderings are currently displayed and need to be synchronized.
- Solution:
  - Synchronization Service
    - Communication Hub between the renderings
    - A rendering only notifies the sync service about its changes. It does not need to be aware of other renderings.
    - A rendering only receives change event from a sync service. It does not need to know who to listen for change events from.

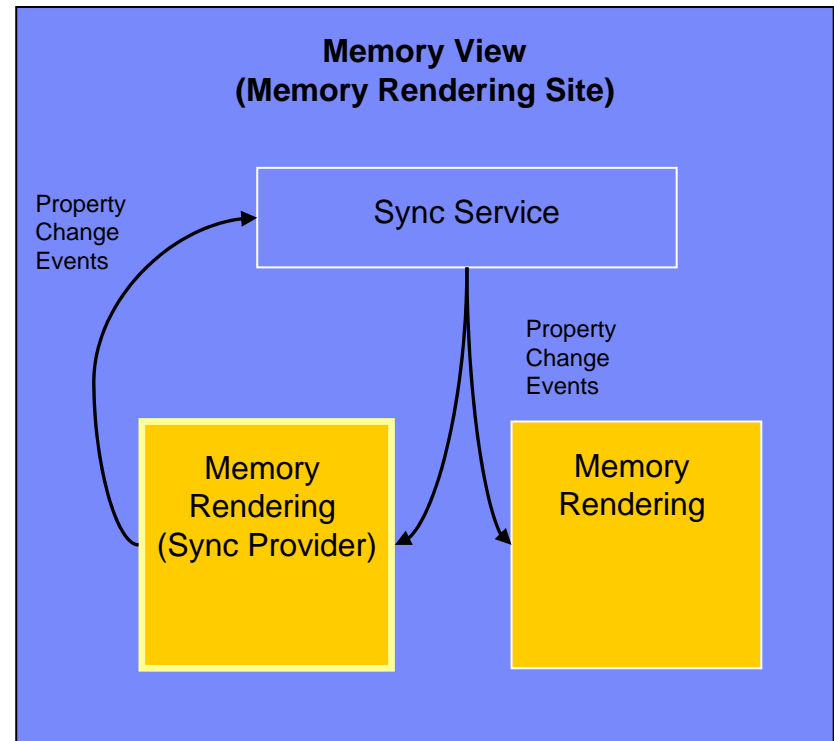
# Synchronization – Setup

- Renderings are activated / deactivated as they gain / lose focus.
- Active Rendering = Synchronization Provider
- Sync Service becomes a property change listener to the synchronization provider.
- Renderings is also a property changed listener to the sync service.
- A rendering defines a list of properties that can be synchronized.
- AbstractMemoryRendering provides this set up for free.



## Synchronization - Interactions

- When one of the properties changed in a rendering, the rendering fires a property changed event.
- The sync service will only receive events from the sync provider. (i.e. the active rendering)
- The sync service notifies all renderings that are interested in the event via a property changed event. Typically this is all renderings showing the same memory block.
- Other renderings receive the event and respond accordingly.



# Checklist

1. Define a list of properties that you wish to synchronize.
2. Implement `IPropertyChangeListener` in your rendering. Your listener should respond to property changed events for the properties that you are interested in.
3. Become a property changed listener to the synchronization service.
4. When a synchronization property is changed in your rendering, fire a property change event.

## Step 1 – Define Synchronization Properties

- AbstractAsyncTableRendering defines a list of properties that can be synchronized:
  - Top Visible Address – changed whenever the user scrolls the rendering.
  - Selected Address – the cursor position of the rendering
  - Row Size – number of addressable units per row.
  - Column Size – number of addressable units per column
  - Page Start Address – for change in the start address of the buffer in a rendering
- Determine which of these properties you wish to synchronize with. Most common ones will be the top visible address / selected address.
- Example:
  - The custom rendering from the exercise has a selected element in the tree. We would like to synchronize this selection with the table rendering. Our custom rendering should handle changes for the selected address property.  
(AbstractAsyncTableRendering #PROPERTY\_SELECTED\_ADDRESS)

## Step 2 – Implement IPropertyChangeListener

- Implement IPropertyChangeListener in your rendering.
  - `public void propertyChange(PropertyChangeEvent event);`
- Tips: When handling the property change event, make sure to check the event source. If the event comes from the same rendering, the rendering should not handle the event.

- Example:

- From AbstractAsyncTableRendering#propertyChanged(...)
  - #selectedAddressChange will move the table cursor to the correct location.
- ```
if (propertyName.equals(AbstractAsyncTableRendering.PROPERTY_SELECTED_ADDRESS)
    && value instanceof BigInteger)
{
    selectedAddressChanged((BigInteger)value);
}
```

## Step 3– Become a listener to Sync Service

- To receive event from the synchronization service, the rendering needs to become a property change event listener from the synchronization service.

```
private void addRenderingToSyncService()
{
    IMemoryRenderingSynchronizationService syncService =
    getMemoryRenderingContainer().getMemoryRenderingSite().getSynchronizationService();

    if (syncService == null)
        return;

    syncService.addPropertyChangeListener(this, null);
}
```

## Step 4 – Fire Property Change Event

- Track changes of synchronization properties in your rendering. When one of the synchronization properties is changed, fire a property change event with the correct id.
- `AbstractMemoryRendering` keeps track property listeners in a rendering. It also provides the `#firePropertyChangedEvent` method for notifying listeners of property changes.
- Example:
  - In `AbstractAsyncTableRendering`, the rendering keeps track of movements in the table cursor. When the table cursor is moved, the rendering fires a change event for `AbstractAyncTableRendering#PROPERTY_SELECTED_ADDRESS`.

```
private void updateSyncSelectedAddress(BigInteger address) {  
    PropertyChangeEvent event = new PropertyChangeEvent(this,  
        AbstractAsyncTableRendering.PROPERTY_SELECTED_ADDRESS, null, address);  
    firePropertyChangedEvent(event);  
}
```

## Exercise 3

- Synchronize the selection in our custom rendering with the table renderings provided by the platform.

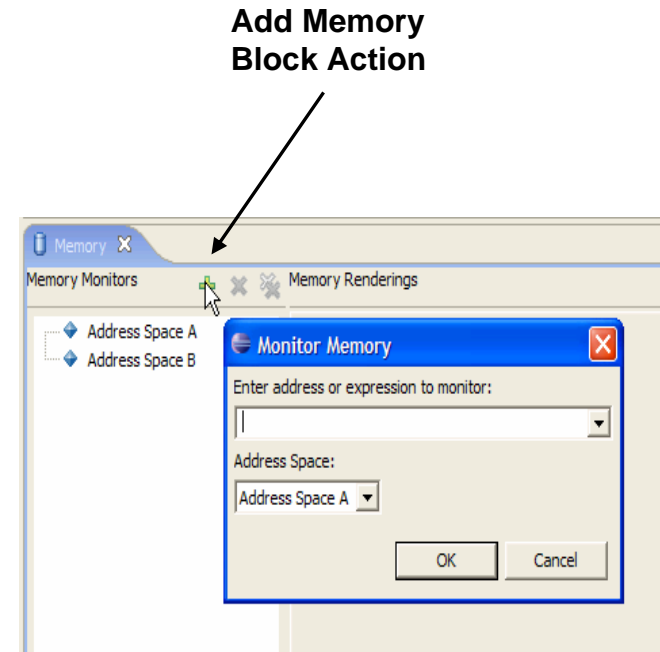
## Advanced Features - IMemoryRenderingBindingProvider

- IMemoryRenderingBindingProvider
  - Allow clients to provide memory rendering binding dynamically.
  - “class” implement IMemoryRenderingBindingsProvider

```
<renderingBindings  
  class="my.memoryrenderingbinding.MemoryBindingProvider">  
  <enablement>  
    <instanceof value="my.memory.block" />  
  </enablement>  
</renderingBindings>
```

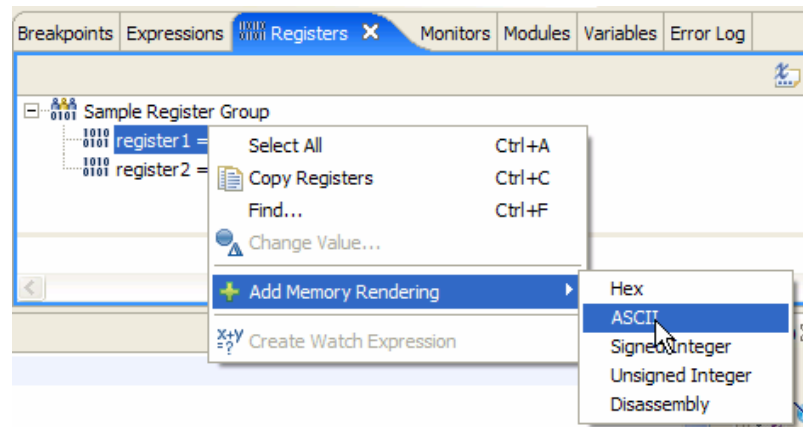
## Advanced Features – Retargettable Action

- Platform provides two retargettable actions for the Memory View
  - Retargettable Add Memory Block Action
  - Retargettable Add Memory Rendering Action
- `IAddMemoryBlocksTarget`
  - Allow clients to override the “add memory block” action from the Memory View. Clients may customize the resulting dialog to prompt user for additional information



# Advanced Features – Retargettable Actions

- IAddMemoryRenderingTarget
  - Platform provides retargettable “Add Memory Rendering” actions for views and editor.
  - Clients may reuse the action to allow user to add memory rendering from other views or from an editor.



# Module 6: Other Enhancements In 3.2

## Other Enhancements: Debug Platform 3.2

- Launch configurations
  - Resource mappings, filtering, and migration support
- Breakpoints
  - Import, export, and ruler actions
- Debug popup dialogs
- Watch expression factory
- Instruction pointer presentation

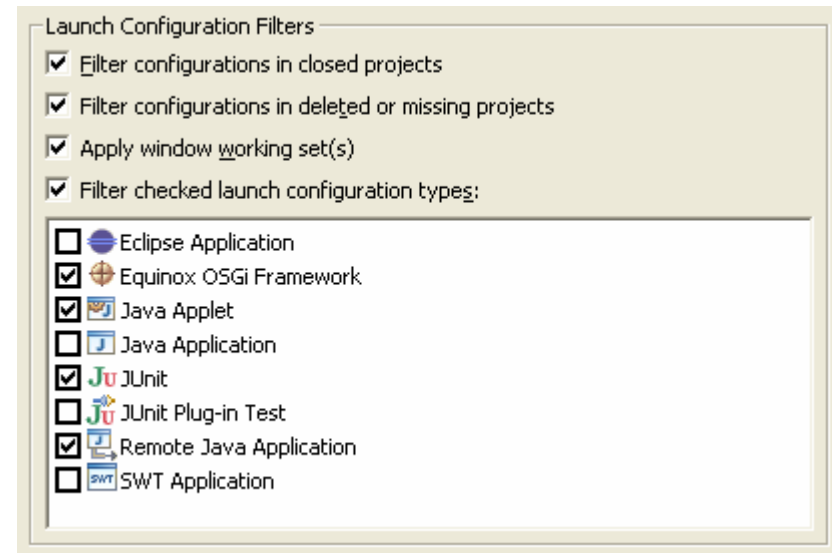
# Launch Configuration Resource Mappings

- Support has been added to associate a set of resources with a launch configuration.
  - Allows the debug UI to perform resource based filtering and cleanup when a project is deleted
  - Clients that contribute a launch configuration type are responsible for maintaining the mapping
  - API has been added to set and get resources:

```
public IResource[] getMappedResources()  
public void setMappedResources(IResource[]  
resources);
```

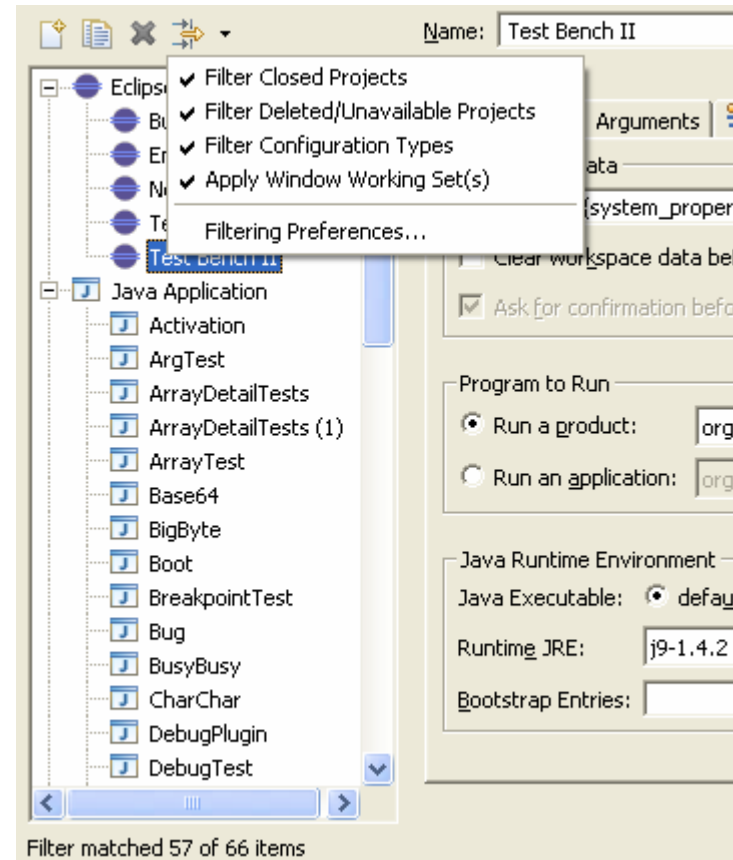
# Launch Configuration Filtering

- Launch configurations are filtered in the launch history and launch dialog based on user preferences
  - Closed projects
  - Unavailable projects
  - Window working sets
  - Specific configuration types



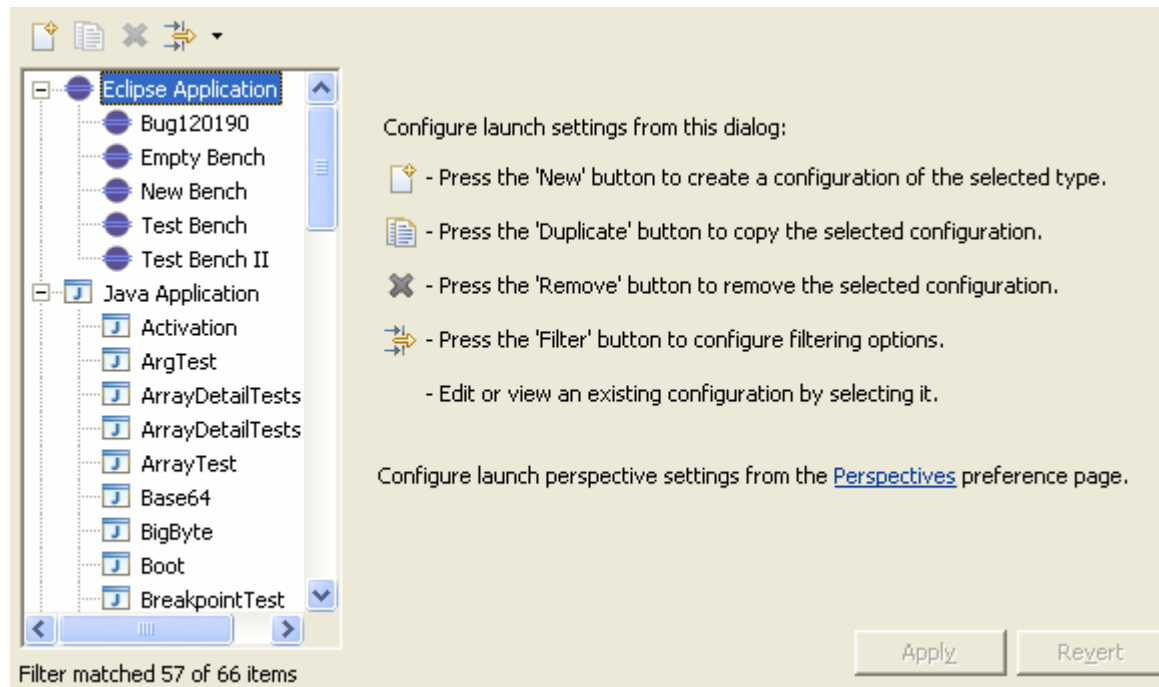
# Launch Configuration Dialog

- A toolbar has been added to the launch dialog to create, delete, and copy configurations, as well as set filtering options.



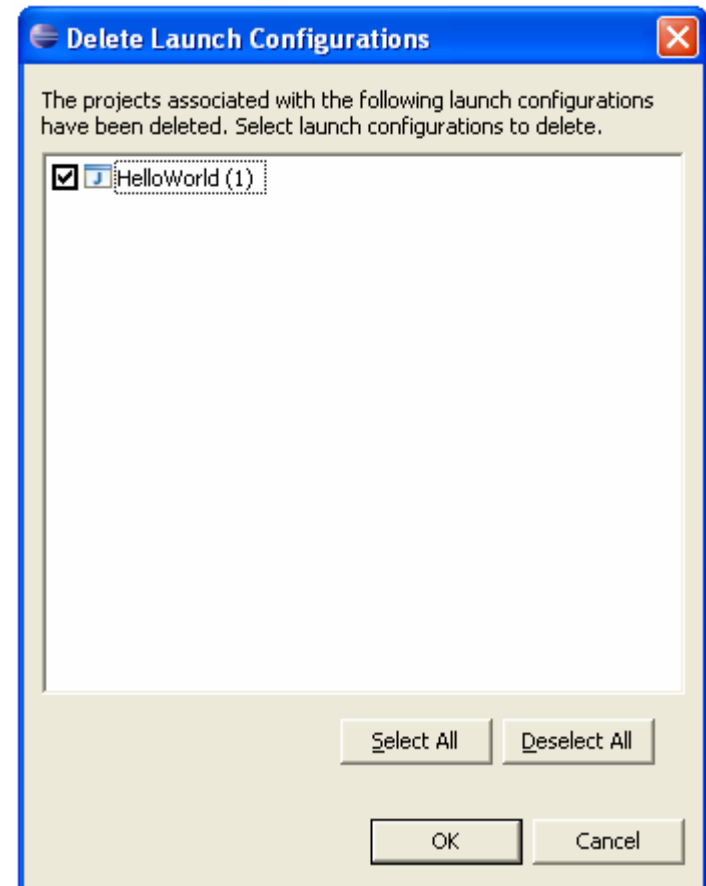
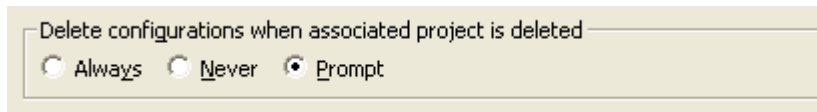
## No more 'Perspectives Tab'

- The perspective settings have been moved to the user preferences, and the dialog has instructions in its place.



# Automatic Cleaning of Launch Configurations

- A user preference controls whether configurations are deleted when their associated project is deleted.



# Launch Configuration Migration

- Support has been added to migrate launch configurations to support new tooling options
  - For example, the new resource mapping feature is a client responsibility and existing configurations must be migrated to leverage the new feature.
  - A migration delegate can be specified by launch configuration types
    - Identifies migration candidates
    - Migrates configurations
  - Users can selectively migrate configurations on the Launch Configurations preference page

# Checklist: Launch Configuration Features

1. Implement `ILaunchConfigurationMigrationDelegate` for your launch configuration types that support resource mappings.

```
public boolean isCandidate(ILaunchConfiguration candidate)
```

```
public void migrate(ILaunchConfiguration candidate)
```

2. Contribute the delegate on the `launchConfigurationType` extension

```
<launchConfigurationType
```

```
    delegate="org.eclipse.jdt.launching.JavaLaunchDelegate"
```

```
    id="org.eclipse.jdt.launching.localJavaApplication"
```

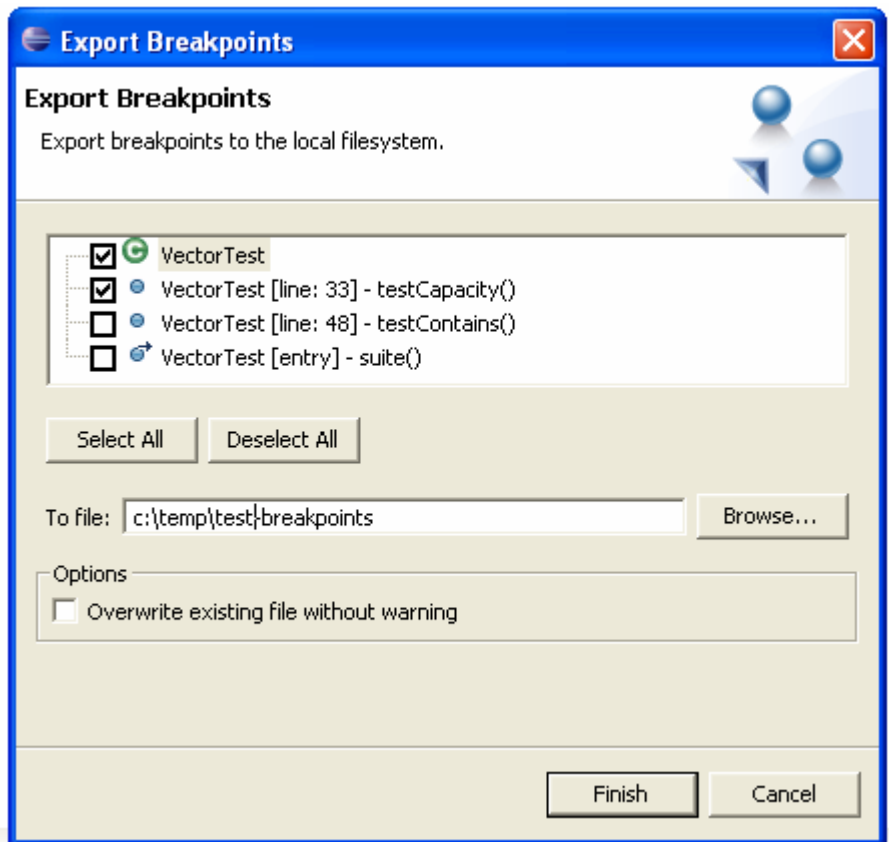
```
    migrationDelegate="org.eclipse.jdt.internal.launching.JavaMigrationDelegate"
```

3. Update any code that creates/modifies launch configurations to initialize/maintain the resource mapping

- Launch shortcuts
- Launch tabs
- Refactoring participants

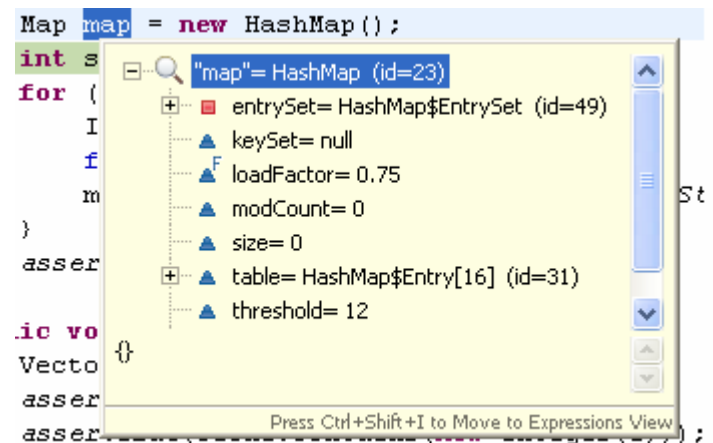
# Breakpoint Import/Export

- Support has been added to export and import breakpoints to/from a file
- Operations have been added as API so you can programmatically import and export breakpoints:
  - `ImportBreakpointsOperation`
  - `ExportBreakpointsOperation`.



# Debug Popup Dialogs

- Popup dialogs have been added as API to support
  - An abstract popup dialog that can be dismissed with a command (key binding) - `DebugPopup`. The Java debugger uses this to “persist” the results in a popup dialog.
  - A popup dialog that displays an expression - `InspectPopupDialog`. The Java debugger uses this class for “Inspecting” expression evaluations.



# Watch Expression Factory

- Support has been added to allow variables to specify a watch expression that should be created for it when the “Create Watch Expression” action is invoked
  - The platform used to use the simple variable name
  - Now the platform delegates to the watch expression factory adapter when provided

```
public interface IWatchExpressionFactoryAdapter {  
public String createWatchExpression(IVariable  
    variable);  
}
```

# 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 (1<sup>st</sup> option), the annotation provides its own text

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

# 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.