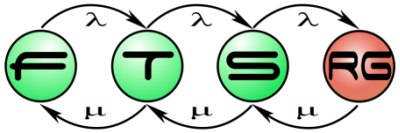
Introduction to Plug-in Development

Eclipse Based Technologies

http://www.inf.mit.bme.hu/edu/courses/eat



Eclipse: Open IDE and Platform

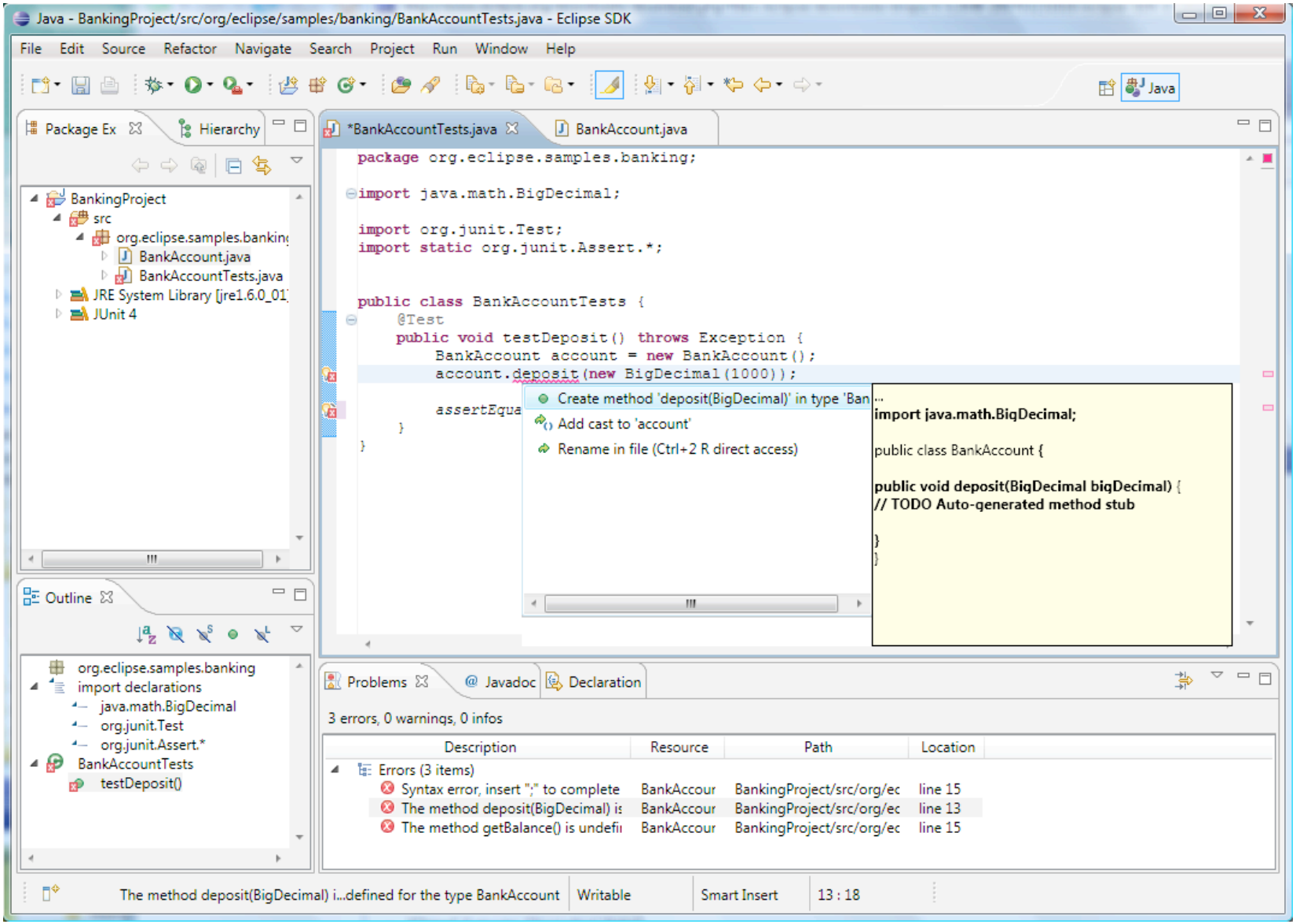
Why Eclipse?

- Strong industrial support
  - IBM, BEA, Oracle (partially)…
- Many successful development
  - Both industrial and research
- Well-designed architecture
  - Modular design (OSGi)
  - Application of design patterns

Why Eclipse?

- Integrated Development Environment (IDE)
  - Multiple languages (Java, C/C++, PHP, …)
  - Graphical editors (pl. UML editor)
  - Connectors to issue trackers, …
- Extensible application platform
  - Scales for very different environments
- Open source

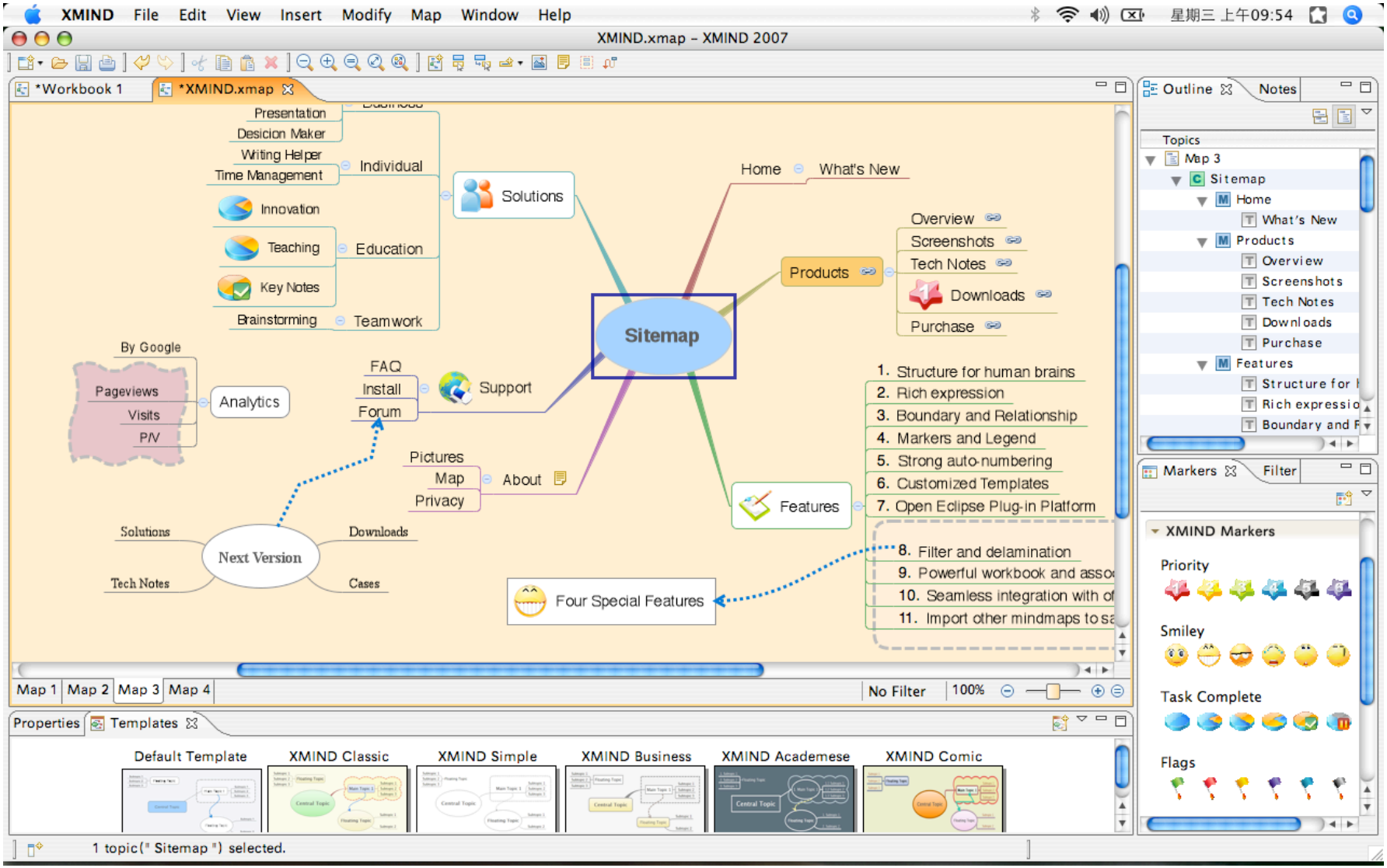
Eclipse: Development Environment



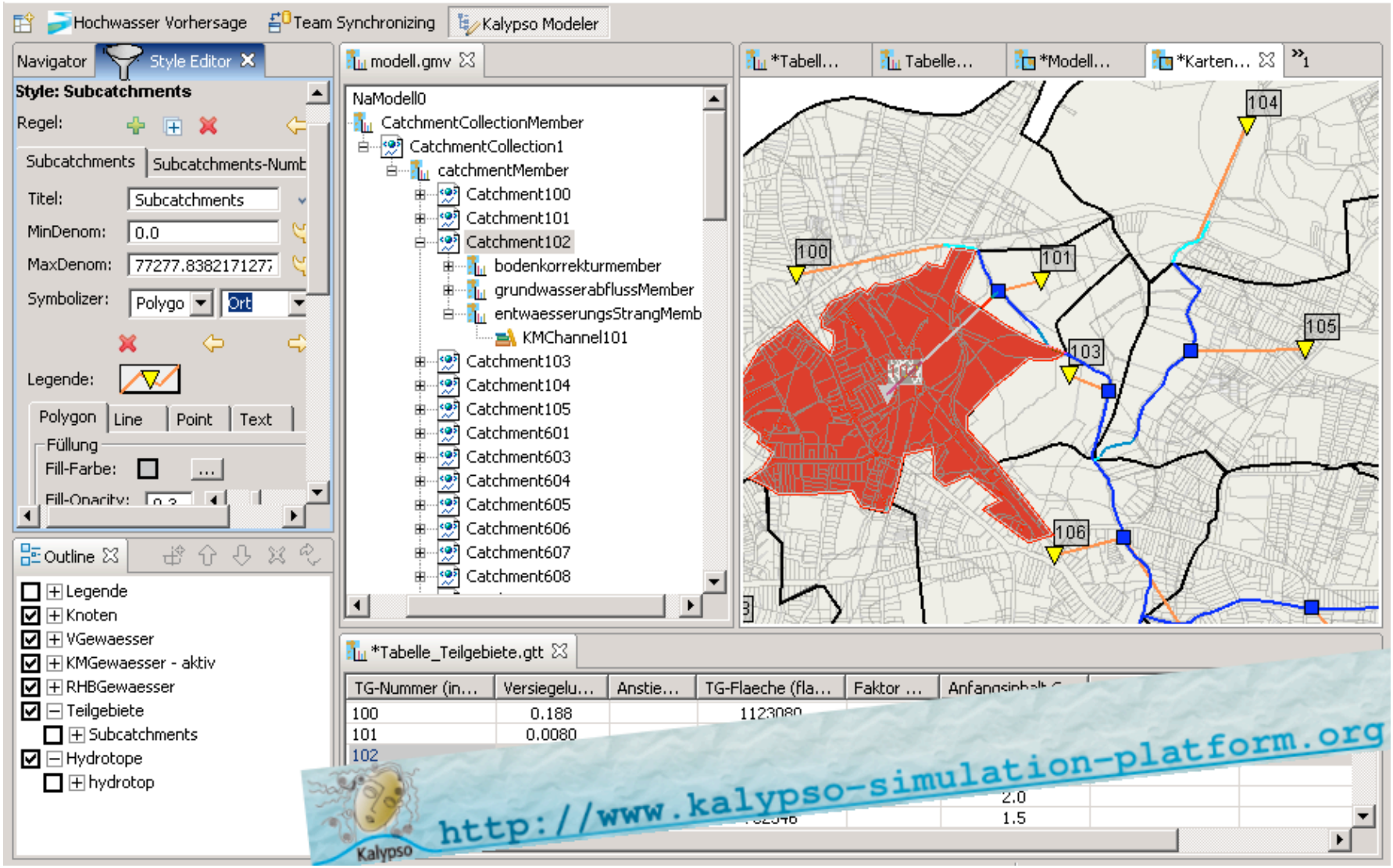
Eclipse: Application Platform

- Rich Client Platform (RCP)
  - Desktop application platform
  - Same extension mechanism as the IDE
    - IDE plug-ins reusable
    - In fact, Eclipse IDE is a specific RCP application
  - Framework for application integration
    - E.g., Lotus Symphony office software
- Rich Ajax Platform (RAP)
  - Web application platform
  - Goal: Single sourcing with RCP

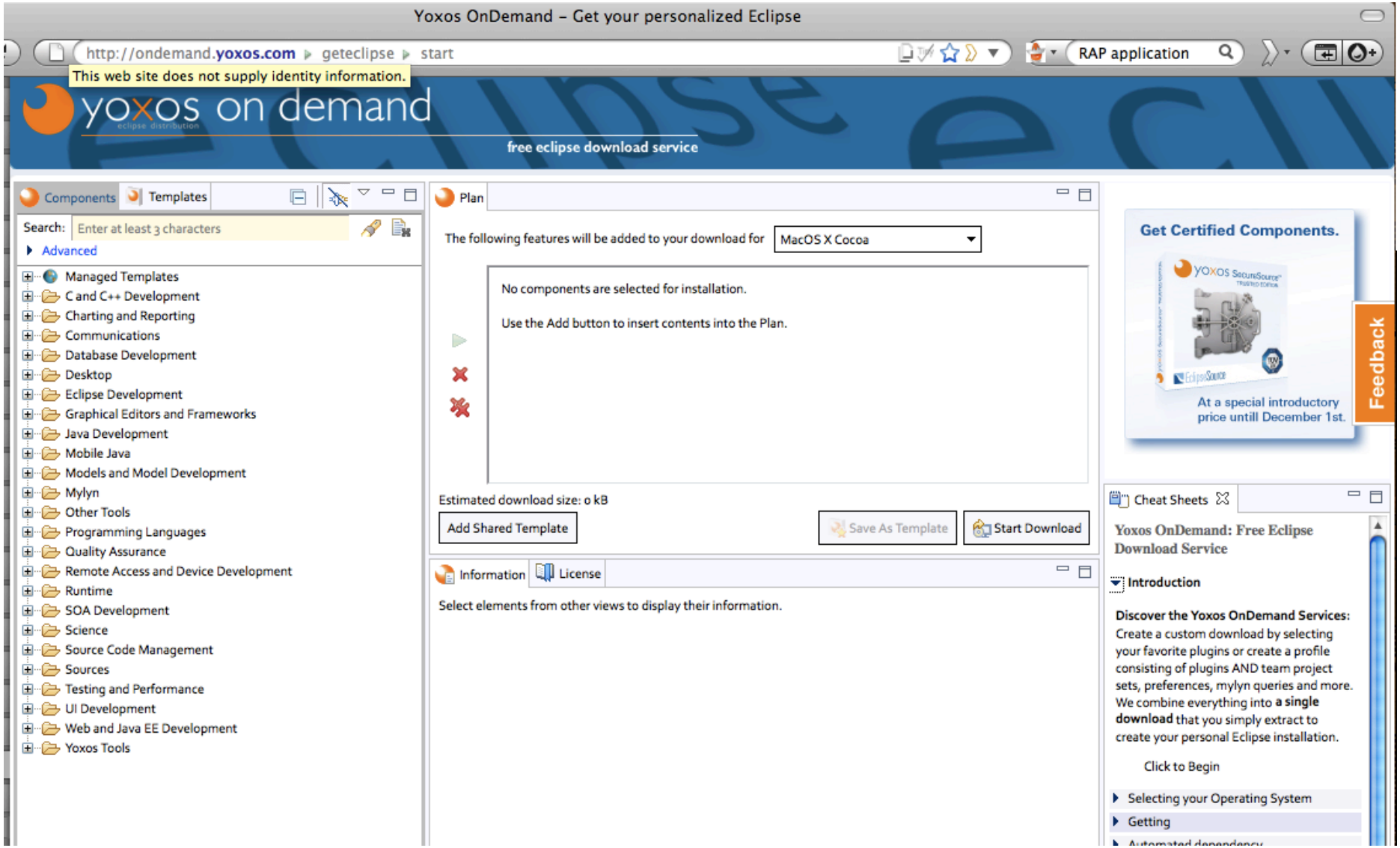
RCP application - XMIND



RCP application- Kalypso

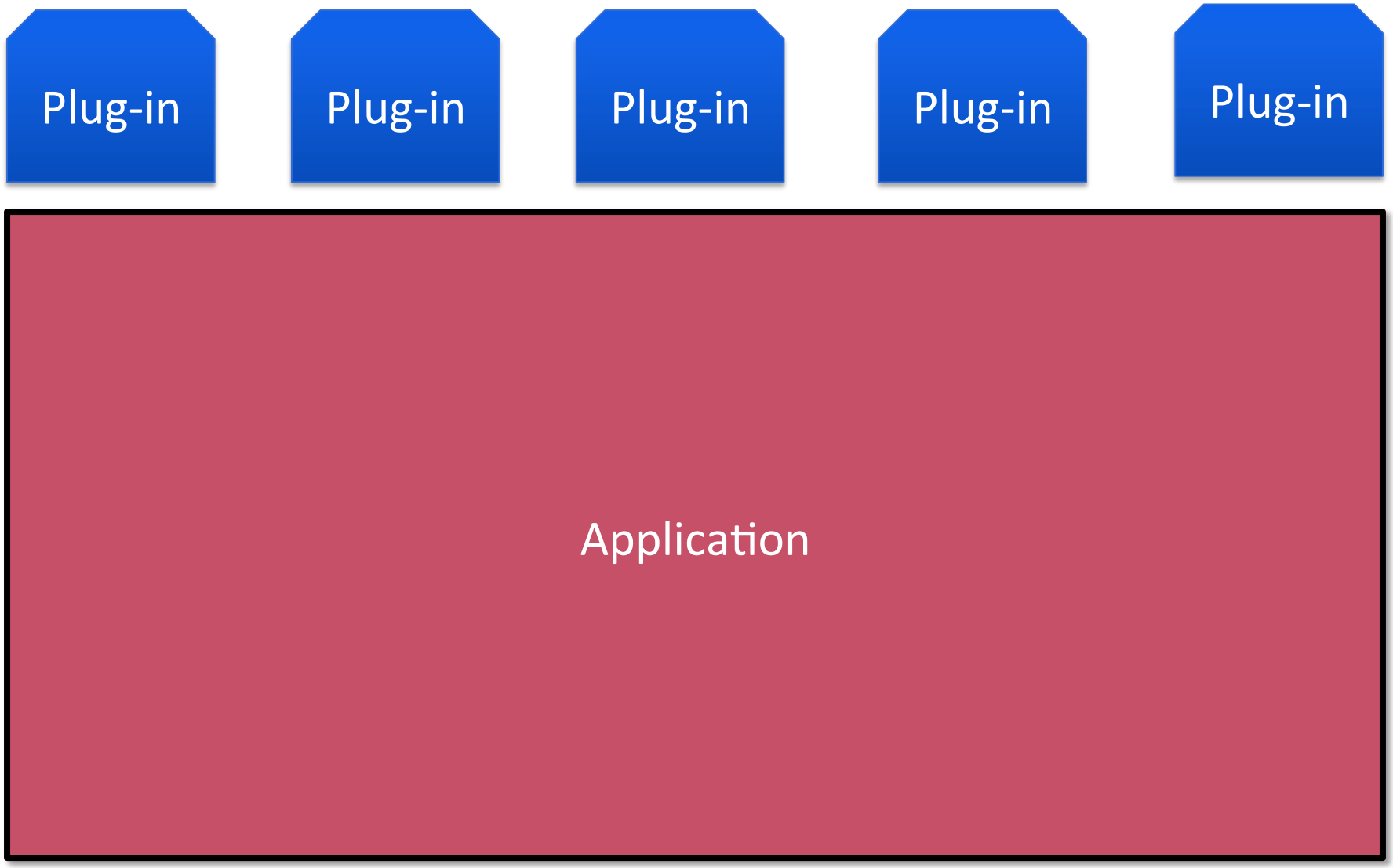


RAP application– Yoxos OnDemand

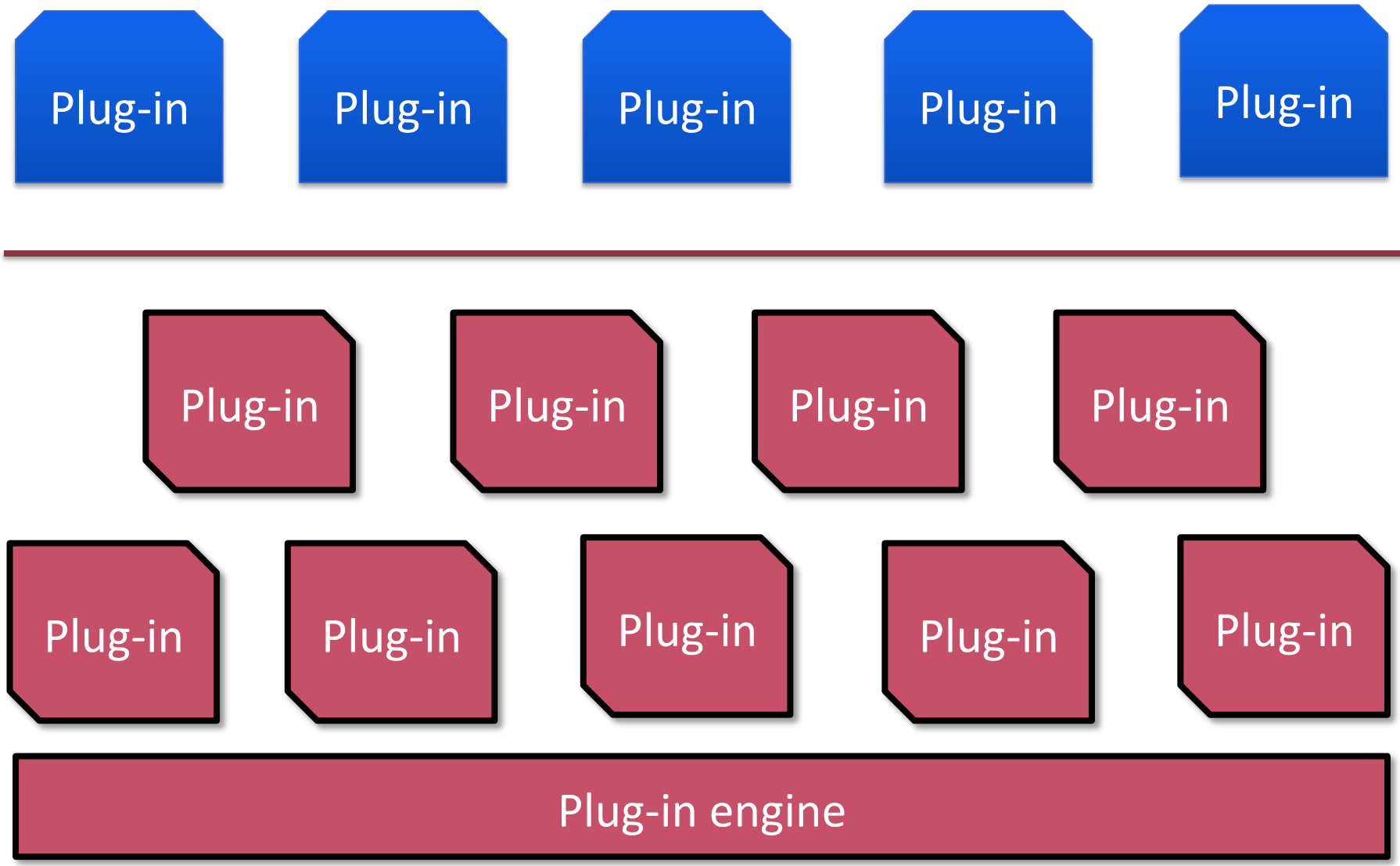


The Plug-in Architecture of Eclipse

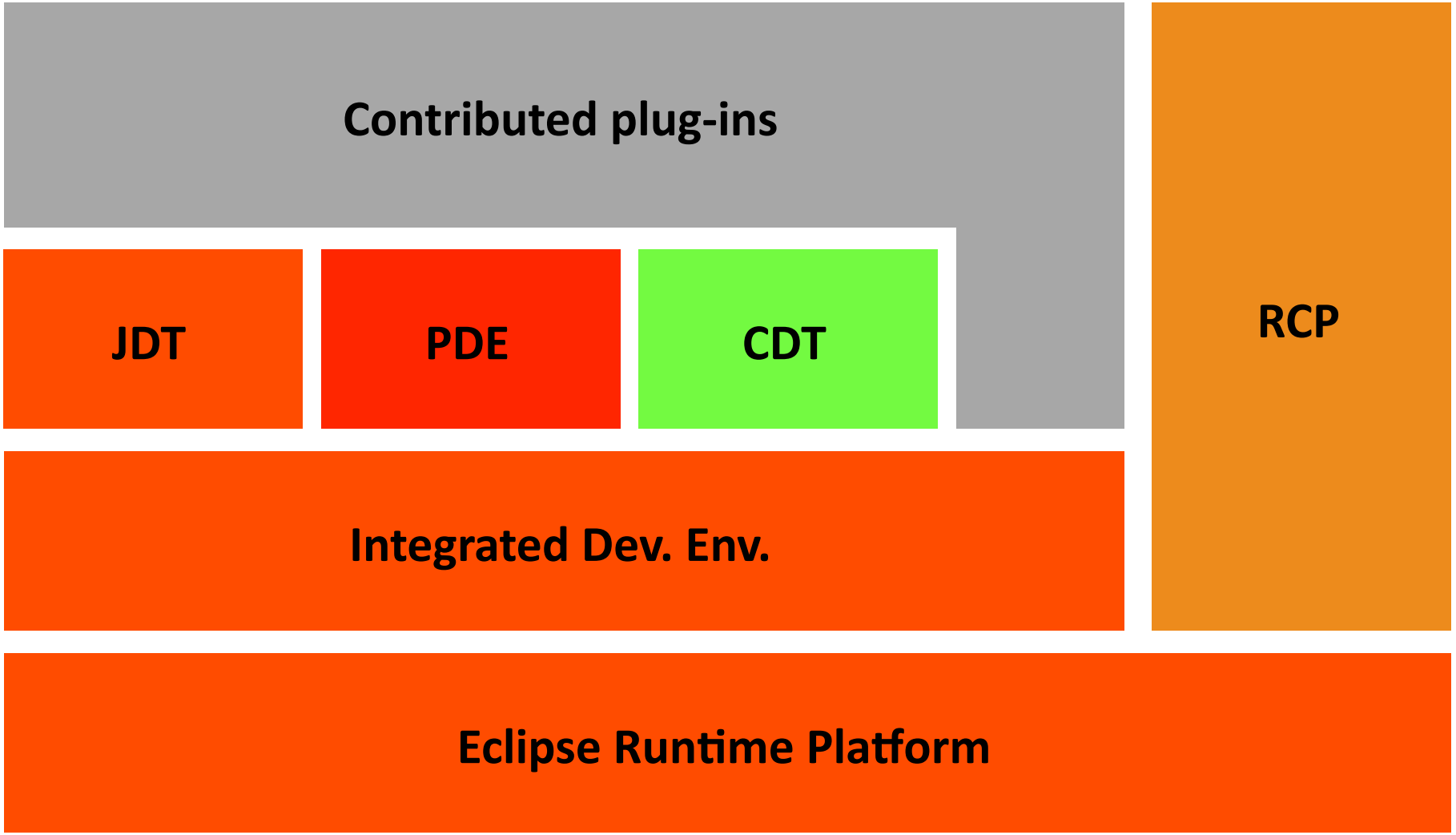
Extensible Application



Plug-in Based Application

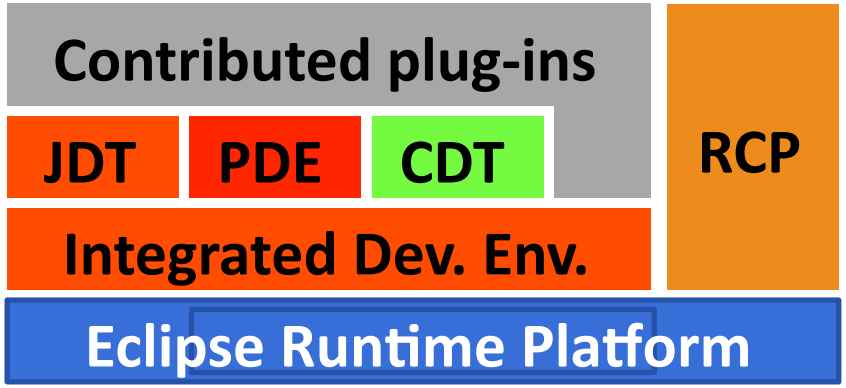


Eclipse Architecture



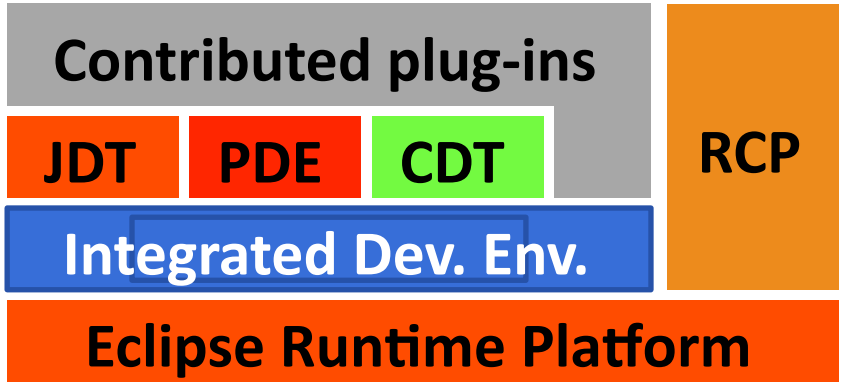
Eclipse Runtime Platform

- Plug-in engine
  - Platform startup
  - Plug-in registry
  - Resource management
- Software updates
- User interface
- Help engine



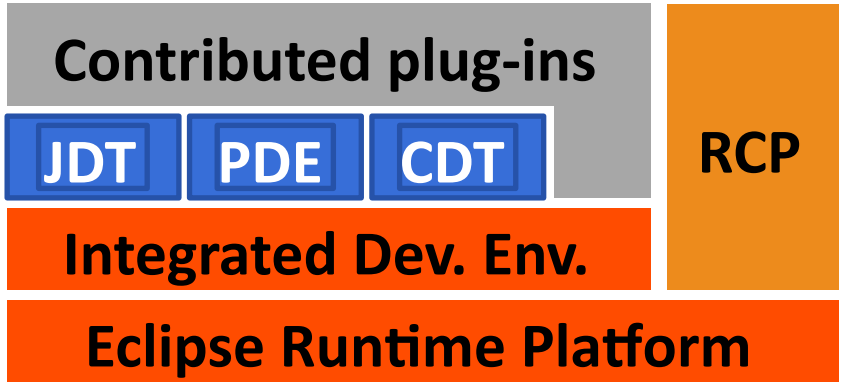
Language-Independent IDE

- Views, perspectives (see later)
- (Language-independent) Search Support
- (Language-independent) Debug Support
- Teamwork support



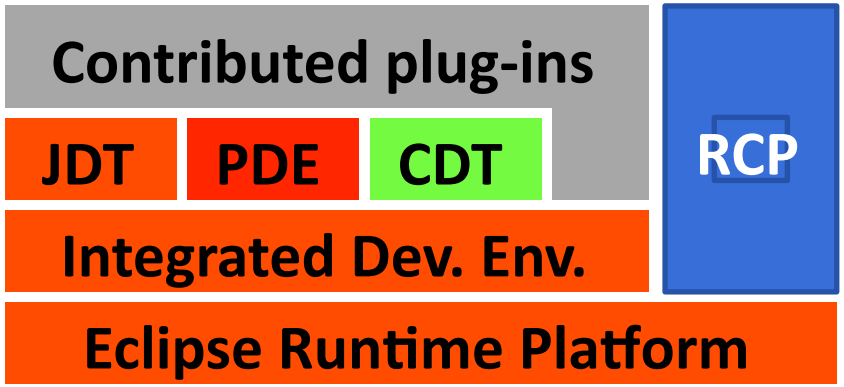
Language-specific Environments

- Java Development Toolkit (JDT)
  - Java Editor, Compiler, Debugger
  - Java Object Model/AST
- Plug-in Development Environment
  - Eclipse plug-in development
- C/C++ Development Tools
  - C/C++ editor
  - External debuggers
  - External compilers



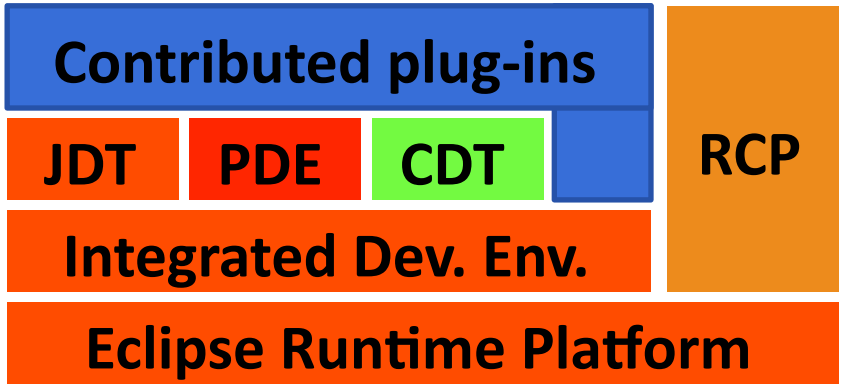
Eclipse Application Platform

- Application Platform
  - Re-useable Services
  - Platform 4.x introduces a new programming model (see later)
- Independent from IDE components
  - But they are re-usable if needed



Contributed Plug-ins

- Everything is possible (almost)
- Can re-use anything
  - Beware of dependencies!



Eclipse Plug-ins

Eclipse Plug-ins (Overview)

- Module concepts
  - Eclipse Plug-ins
  - *OSGi bundles (later in this course)*
- Co-operation
  - Calling API methods
  - Extension points
  - *OSGi services (later in this course)*

Eclipse Plug-in

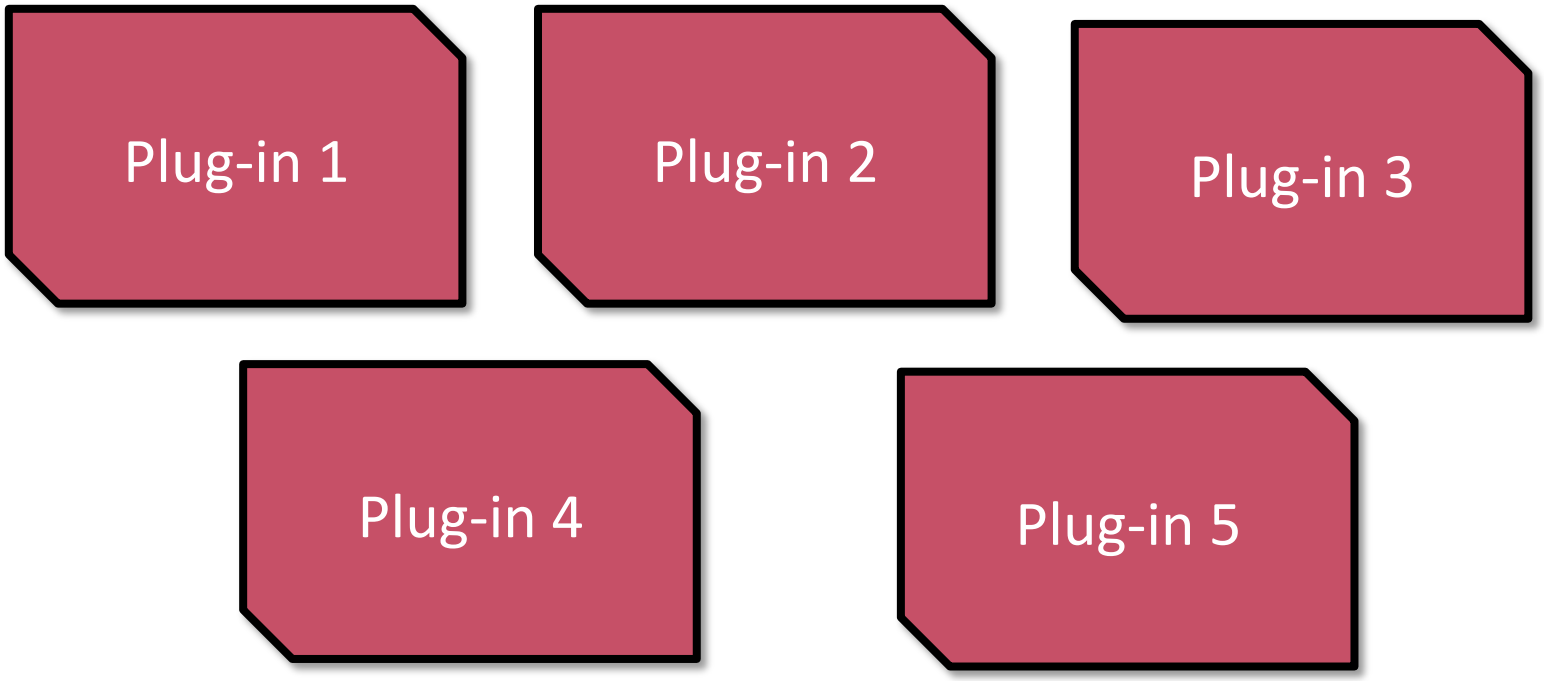
- Smallest installable unit (sort of)
  - Java code
  - Metadata (descriptors)
- Packaging/distribution
  - jar archives
  - Recompiling existing libraries to plug-ins manageable
- Extension
  - Features: a group of plug-ins
  - Used at install (why?)

Lazy loading

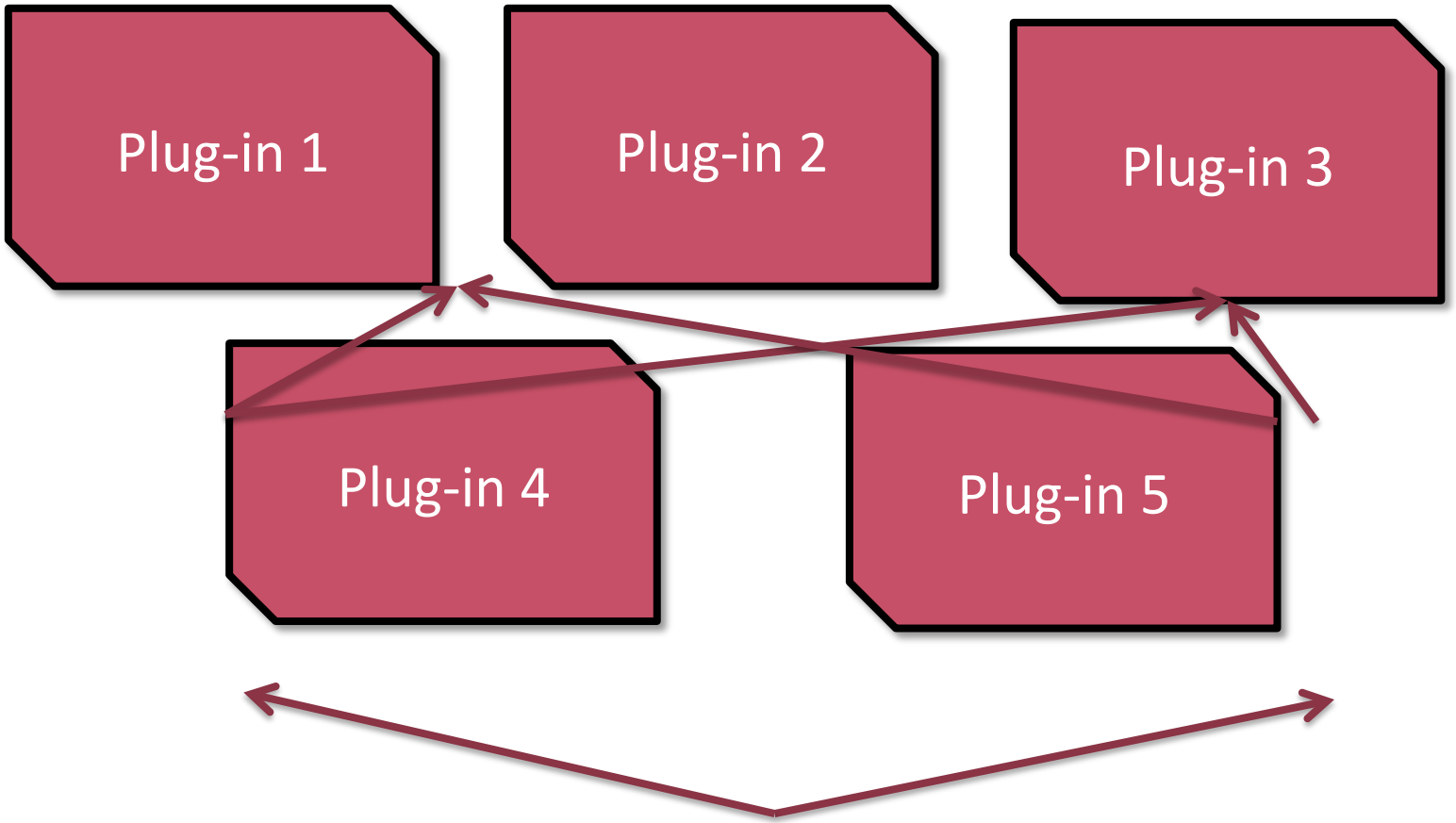
- Basic rule
  - „Contributions are only loaded when they are needed.”
  - BUT: Plug-in metadata is always loaded

Most important metadata: Dependencies

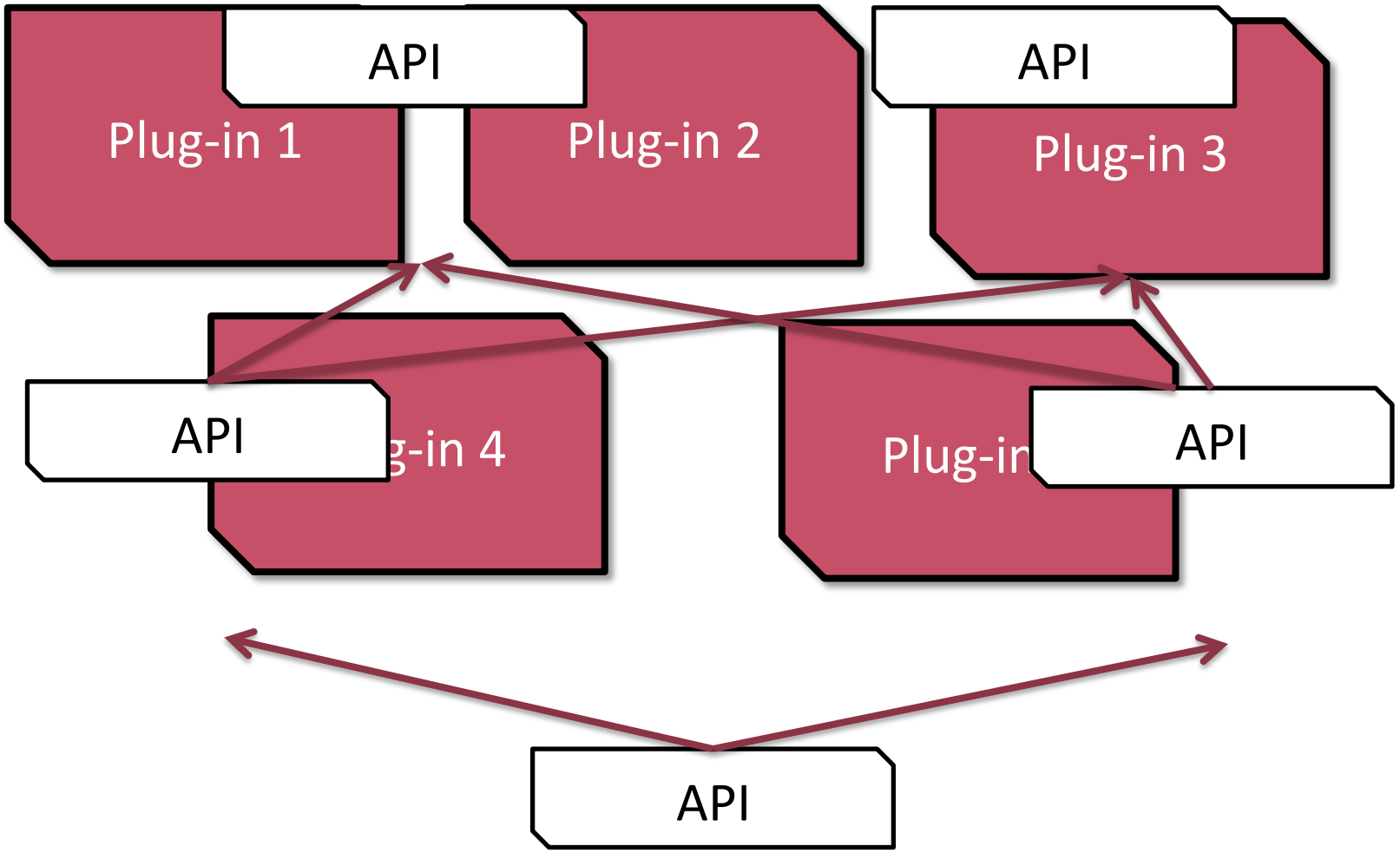
Most important metadata: Dependencies



Most important metadata: Dependencies



Most important metadata: Dependencies



Standard Project Separation

- Core plug-in
  - Mandates API definition
- GUI plug-in
  - Allows headless execution if necessary
- Further plug-ins as necessary
  - In most cases, dependencies define boundaries
  - Avoid large plug-ins

API Definition

- List API packages (exported)
  - By default, packages are visible only inside the plug-in
  - Exported packages are also visible in other plug-ins
    - Via dependency definition
- Goal
  - Information hiding
  - Implementation classes not needed

API Design – Rules of Thumbs

- Explicit API: *Separate the API from internals*
  - Typically *.internal* packages
- Stability: *Once you invite others, don't change it*
  - API changes -> every user has to change
- Defensive API design:
  - *Reveal only the API in which you have confidence, but be prepared to reveal more API as clients ask for it.*

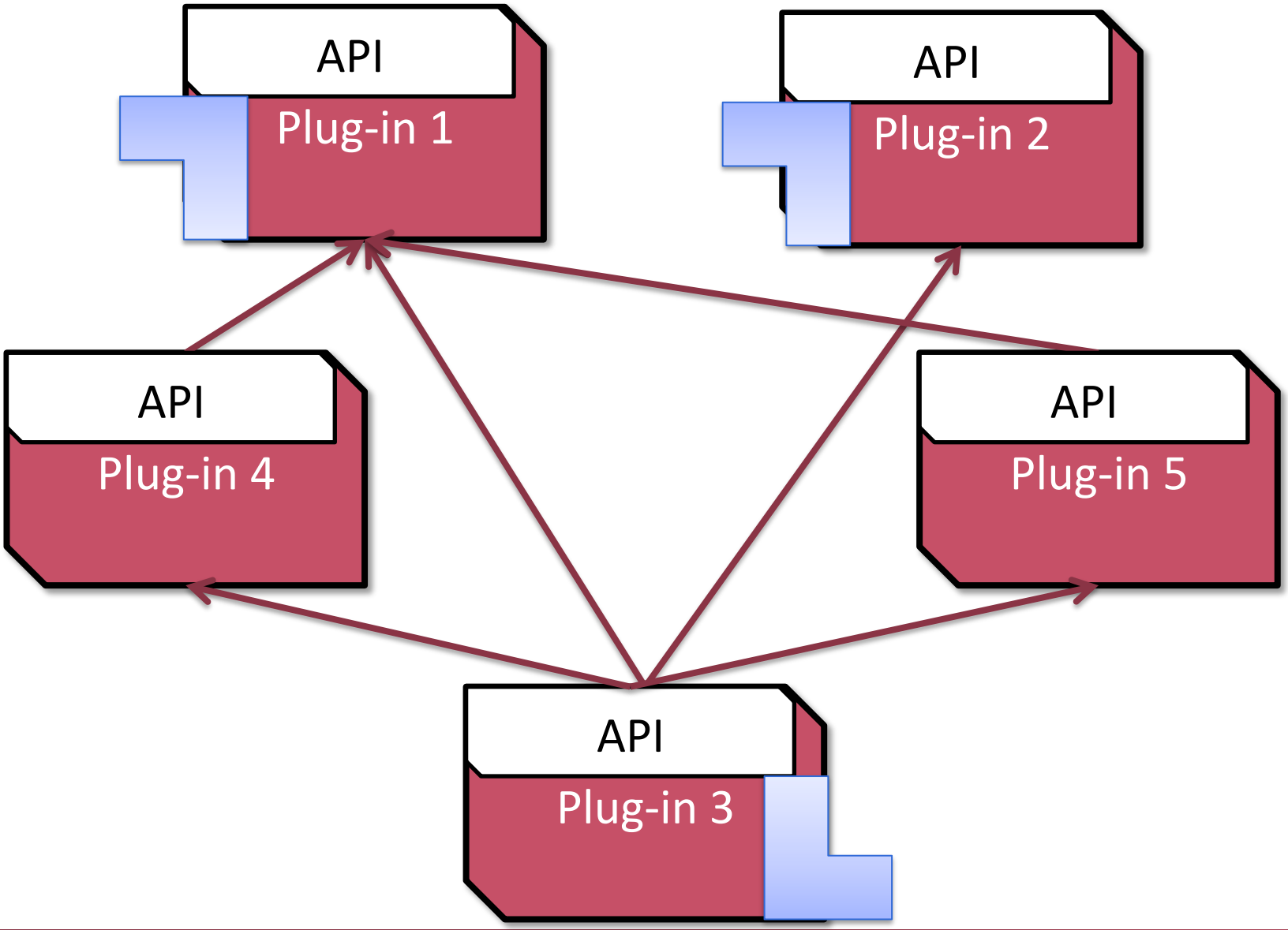
Problem with Generic Plug-ins

- Generic GUI extensions (e.g. menu items)
  - Unique labels and event handlers
    - **Must** be implemented in specific plug-ins
  - Unified presentation and management
    - **Should** be implemented in general plug-ins
- Problem
  - Generic plug-in **must not depend on** specific plug-ins
  - Dependency hierarchy/circles
- **A** solution
  - Extension points

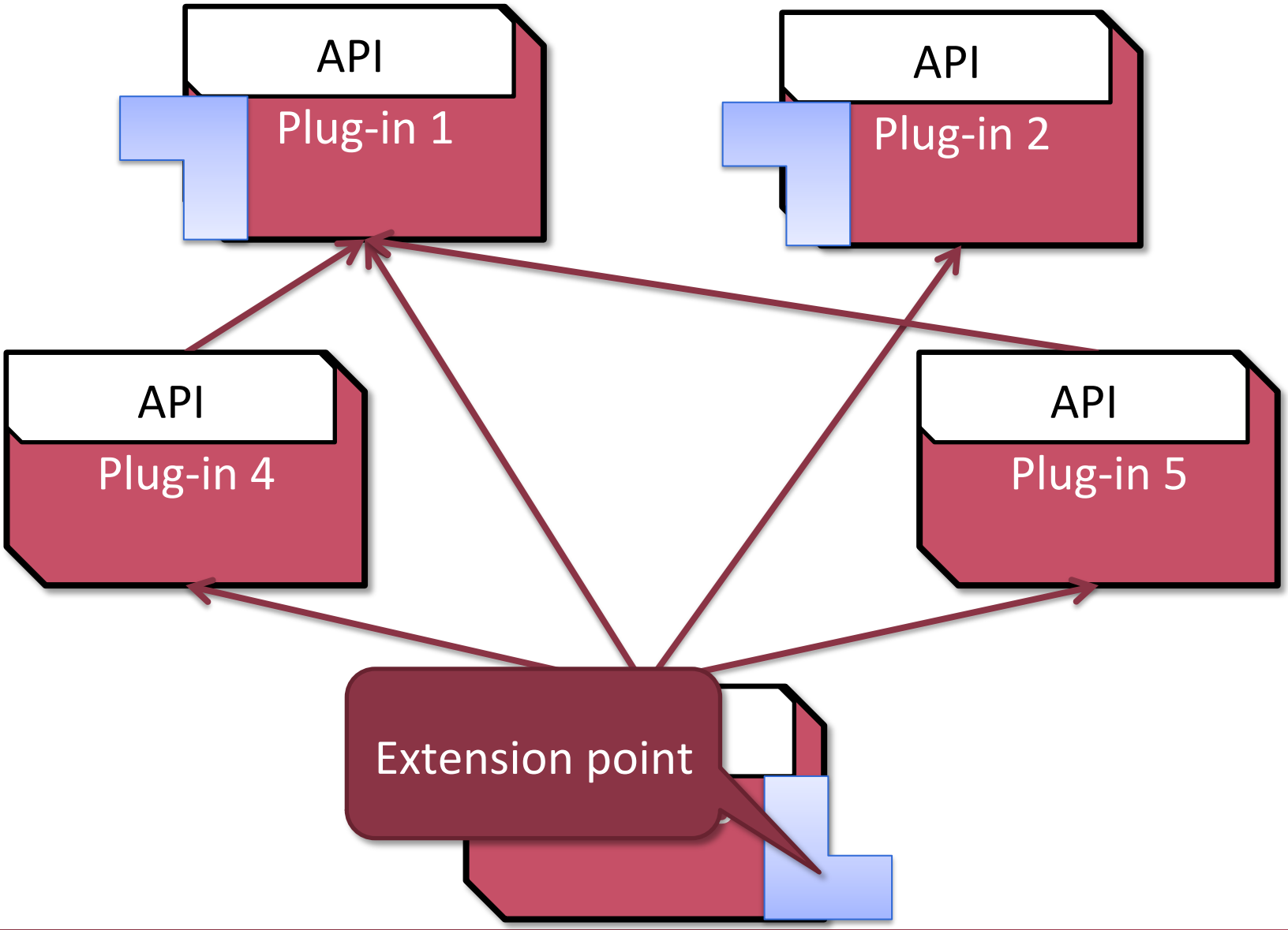
Extension Points

- Eclipse extension point
  - Definer: general plug-in
    - Extension point schema
  - Users: specific plug-ins
    - Configuration
    - Java code
    - Corresponds to schema
      - Dependency to the general plug-in
- Can instantiate non-API classes!

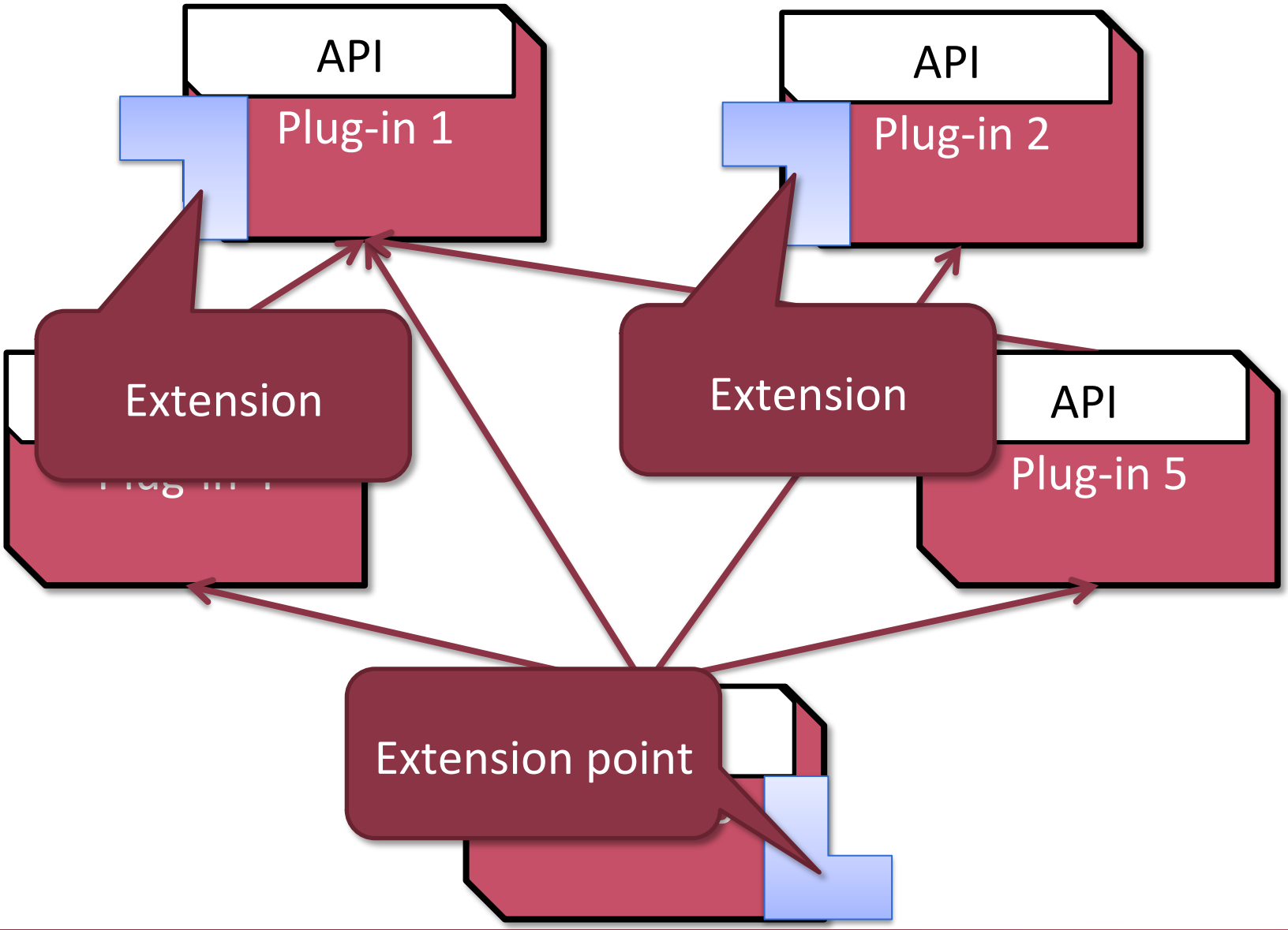
Extension point



Extension point



Extension point



Extension point – Rules of Thumb

- Error management
  - Do not assume correct extensions!
  - *As the provider of an extension point, you must protect yourself against misbehavior on the part of extenders*
- Invitation rule
  - *Whenever possible, let others contribute to your contributions.*
- Documentation
  - Always document the extension points

Extensions – Rules of Thumb

- *Contributions must conform to expected interfaces*
  - Extension must match the extension point definition
- *Share, don't replace*
  - An extension must work with other extensions
    - Do **not** try to exclude other extensions
    - Do not assume there are **no other extensions**
  - Exception:
    - Some extension point assume a single extension
      - Pl. org.eclipse.core.resources.teamHook
    - Extension point documentation!

Extension point - Implementation

- Using extensions
  1. **IExtensionRegistry instance:**
     `Platform.getExtensionRegistry()`
  2. **IExtensionPoint instance:**
     `registry.getExtensionPoint(String)`
  3. **IExtension[] instances:**
     `point.getExtensions()`
  4. **IConfigurationElement instance:**
     `extension.getConfigurationElements()`

IConfigurationElement

- Extension parameters
  - Tree structure (like the xml descriptor)
  - Can be validated (extension point schema)

- `String element.getAttribute(String)`
  - Attribute name as parameter
- `Object element.createExecutableExtension(String)`
  - Instantiating classes
    - Can instantiate non-exported classes as well
  - Only parameterless constructors can be used
  - Type casting required

Plug-in Rules of Thumb

- *"Monkey see/monkey do"*
  - *Always copy the structure of a similar plug-in*
    - Template wizards in Eclipse
- Relevance rule
  - *Contribute only when you can successfully operate*
  - Be careful to avoid cluttered user interface
- Follow platform conventions!
  - Re-use existing extensions/services
  - Copy the user interface (if possible)
  - User Interface Guidelines:
    - http://wiki.eclipse.org/User_Interface_Guidelines

Further Reading about Compatibility

- API Design and Evolution
  - http://www.slideshare.net/moberhuber/eclipsecon-2010-api-design-and-evolution-tutorial
- Evolving Java-based APIs
  - http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs

Plug-in projects

Plug-in project structure

- Metadata
  - manifest.mf
    - Name, version numbers and dependencies
  - plugin.xml
    - Optional
    - Extension and extension point definitions
- Implementation
  - Java code
- Resources
  - E.g., images, icons, configuration files

Plug-in project structure

- Metadata
  - manifest.mf
    - Name, version numbers and dependencies
  - plugin.xml
    - Optional
    - Extension and extension point definitions
- Implementation
  - Java code
- Resources
  - E.g., images, icons, configuration files

OSGi convention

Plug-in project structure

- Metadata
  - manifest.mf
    - Name, version numbers and dependencies
  - plugin.xml
    - Optional
    - Extension and extension point definitions
- Implementation
  - Java code
- Resources
  - E.g., images, icons, configuration files

OSGi convention

Eclipse specific

manifest.mf

- Plugin ID (mandatory)
  - Programmatic identification of the plug-in
- Plugin name (mandatory)
  - Name to display on user interface
- Version number (mandatory)
  - Semantic versioning (next slide)
- Dependencies
  - List of plug-ins (with version numbers)

Semantic versioning

- Fixed format: x.y.z.qualifier
  - x: **major** version
    - an increase signals compatibility breaking change
  - y: **minor** version
    - an increase signals backwards compatible changes
  - z: **patch** version
    - a bugfix release
  - **qualifier**
    - Build identifier number
    - Can be generated automatically
- Version comparison:
  - Comparing each version number as a string
- Example:
  - Last Eclipse 3.4 platform release:
    - 3.4.2.R200902111700

plugin.xml

- Extension point definition
  - XSD schema to define possible extensions
    - String literals, Java class or resource references
    - Mandatory/optional element definition
  - Documentation
- Extension definitions
  - Refers extension point
    - From any visible plug-in (including itself)
  - XML description validated by the extension point schema
  - Java class and resource references

plugin.xml example

```
<?eclipse version="3.4"?>
<plugin>
  <extension
        point="org.eclipse.ui.commands">
     <command
          description="Shows a Helloworld message."
          id=”hu.bme.mit.commanddemo.helloworld"
          name="Display hello world!">
     </command>
  </extension>
 <extension
        point="org.eclipse.ui.handlers">
     <handler
          class=”hu.bme.mit.commanddemo.commands.HelloworldHandler"
          commandId=”hu.bme.mit.commanddemo.helloworld">
     </handler>
  </extension>
</plugin>
```

plugin.xml example

```
<?eclipse version="3.4"?>
<plugin>
  <extension
       point="org.eclipse.ui.commands">
    <command
         description="Shows a Helloworld message."
         id=”hu.bme.mit.commanddemo.helloworld"
         name="Display hello world!">
    </command>
  </extension>
 <extension
       point="org.eclipse.ui.handlers">
    <handler
         class=”hu.bme.mit.commanddemo.commands.HelloworldHandler"
         commandId=”hu.bme.mit.commanddemo.helloworld">
    </handler>
  </extension>
</plugin>
```

Extension point definition

plugin.xml example

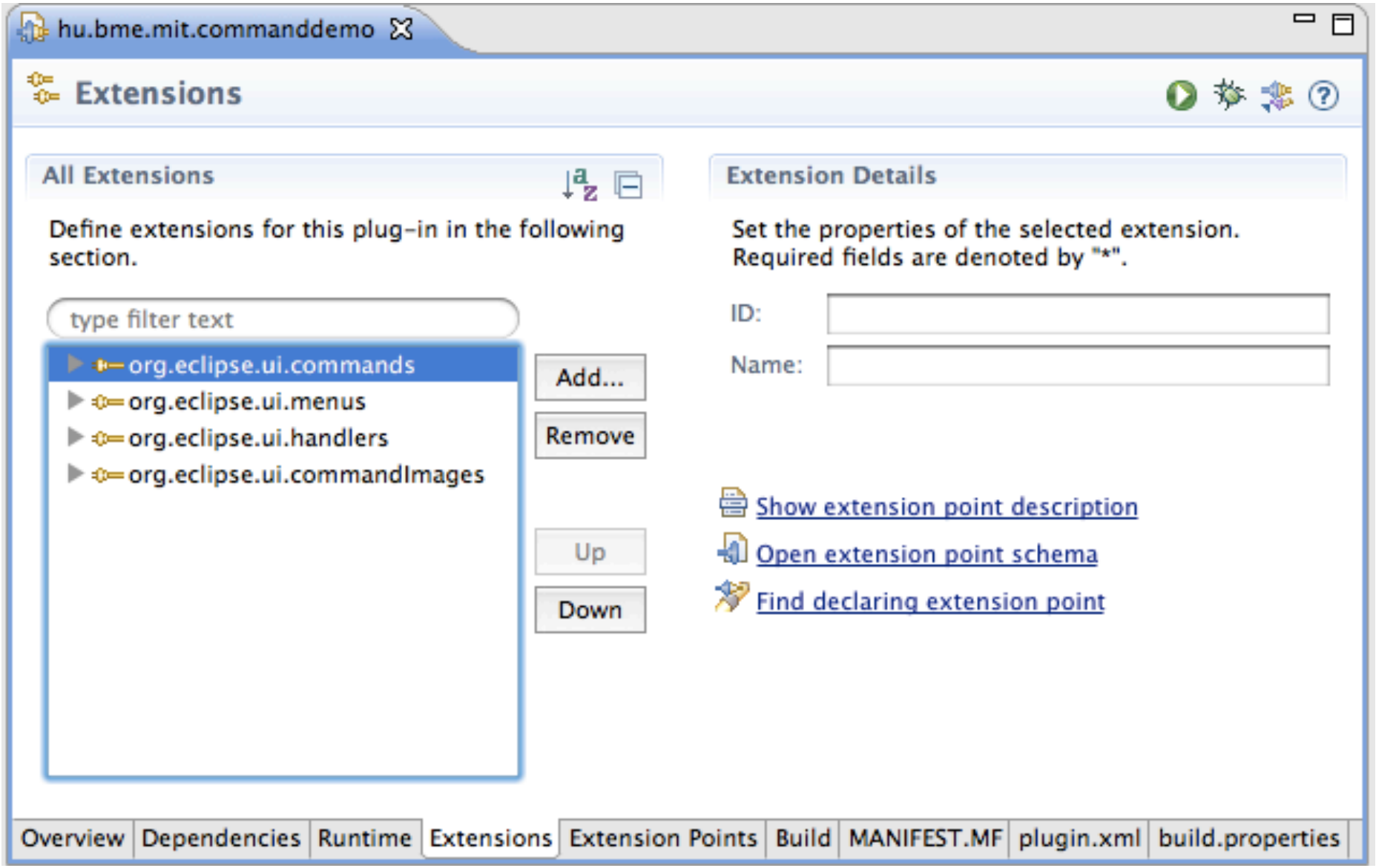
```
<?eclipse version="3.4"?>
<plugin>
  <extension
        point="org.eclipse.ui.commands">
    <command
          description="Shows a Helloworld message."
          id=”hu.bme.mit.commanddemo.helloworld"
          name="Display hello world!">
    </command>
  </extension>
 <extension
        point="org.eclipse.ui.handlers">
    <handler
          class=”hu.bme.mit.commanddemo.commands.HelloworldHandler"
          commandId=”hu.bme.mit.commanddemo.helloworld">
    </handler>
  </extension>
</plugin>
```

plugin.xml example

```
<?eclipse version="3.4"?>
<plugin>
  <extension
        point="org.eclipse.ui.commands">
     <command
          description="Shows a Helloworld message."
          id=”hu.bme.mit.commanddemo.helloworld"
          name="Display hello world!">
     </command>
  </extension>
 <extension
        point="org.eclipse.ui.handlers">
     <handler
          class=”hu.bme.mit.commanddemo.commands.HelloworldHandler"
          commandId=”hu.bme.mit.commanddemo.helloworld">
     </handler>
  </extension>
</plugin>
```

Class reference

Form based metadata editor



Executing Eclipse Plug-ins

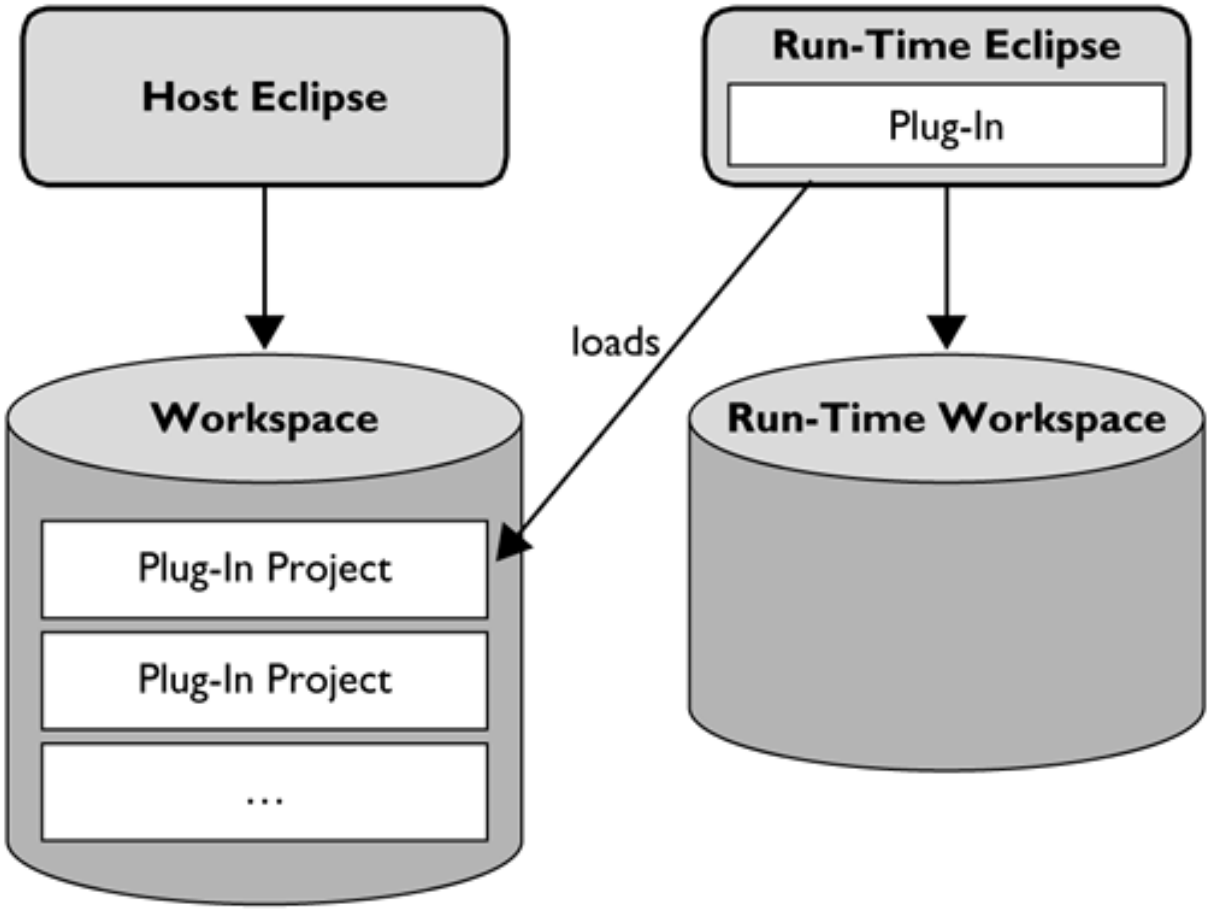
Target Platform

- A set of plug-ins
- Contains the developed plug-ins dependencies for
  - Compiling
  - Execution
- Different platforms for
  - IDE plug-ins
  - RCP applications
  - RAP applications
- Custom platforms possible

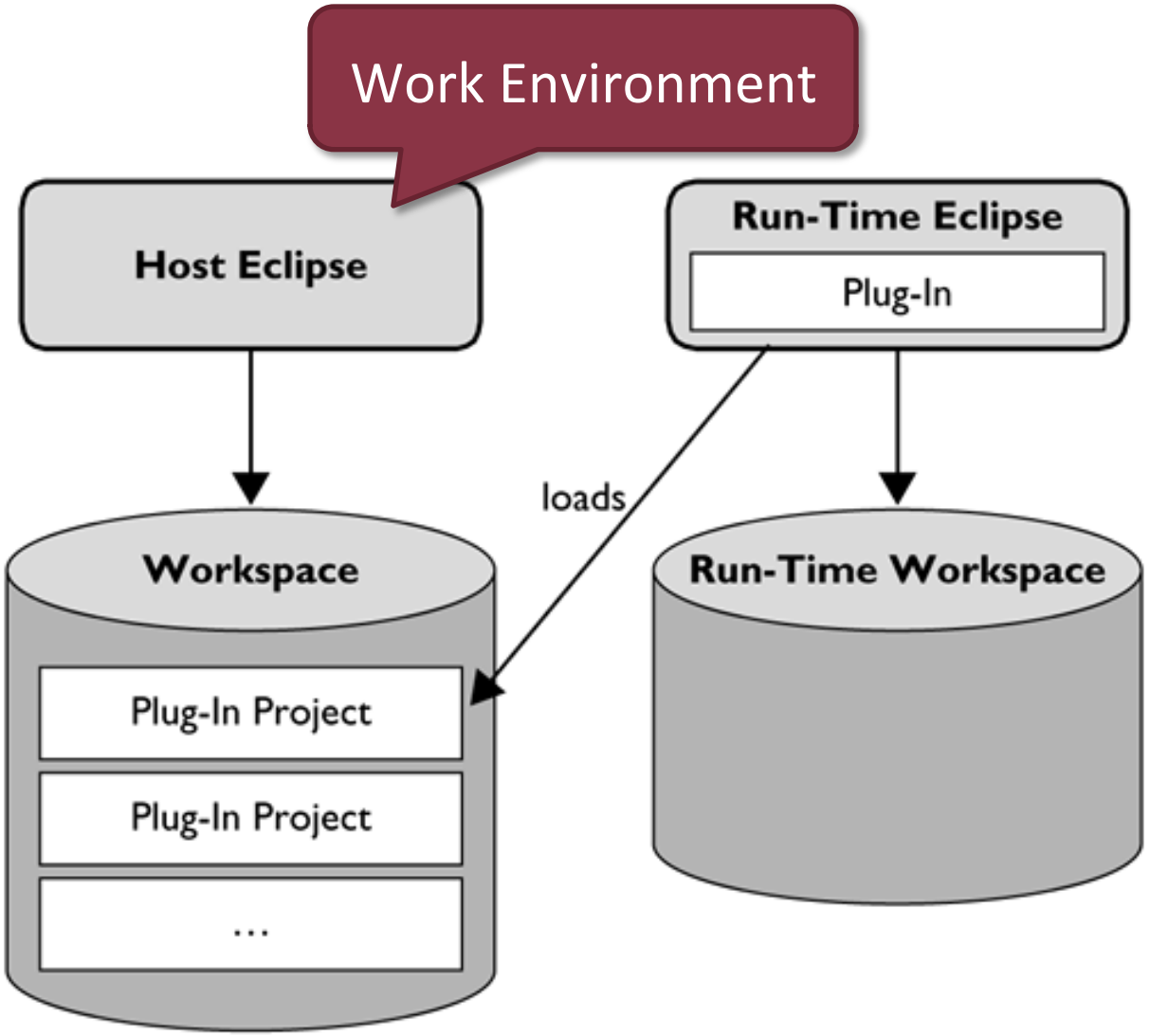
Plug-in Execution

- Execution classpath contains
  - Developed plug-ins
  - Target platform
- IDE plug-in
  - A new Eclipse workbench is started
- RCP/RAP application
  - The defined application starts

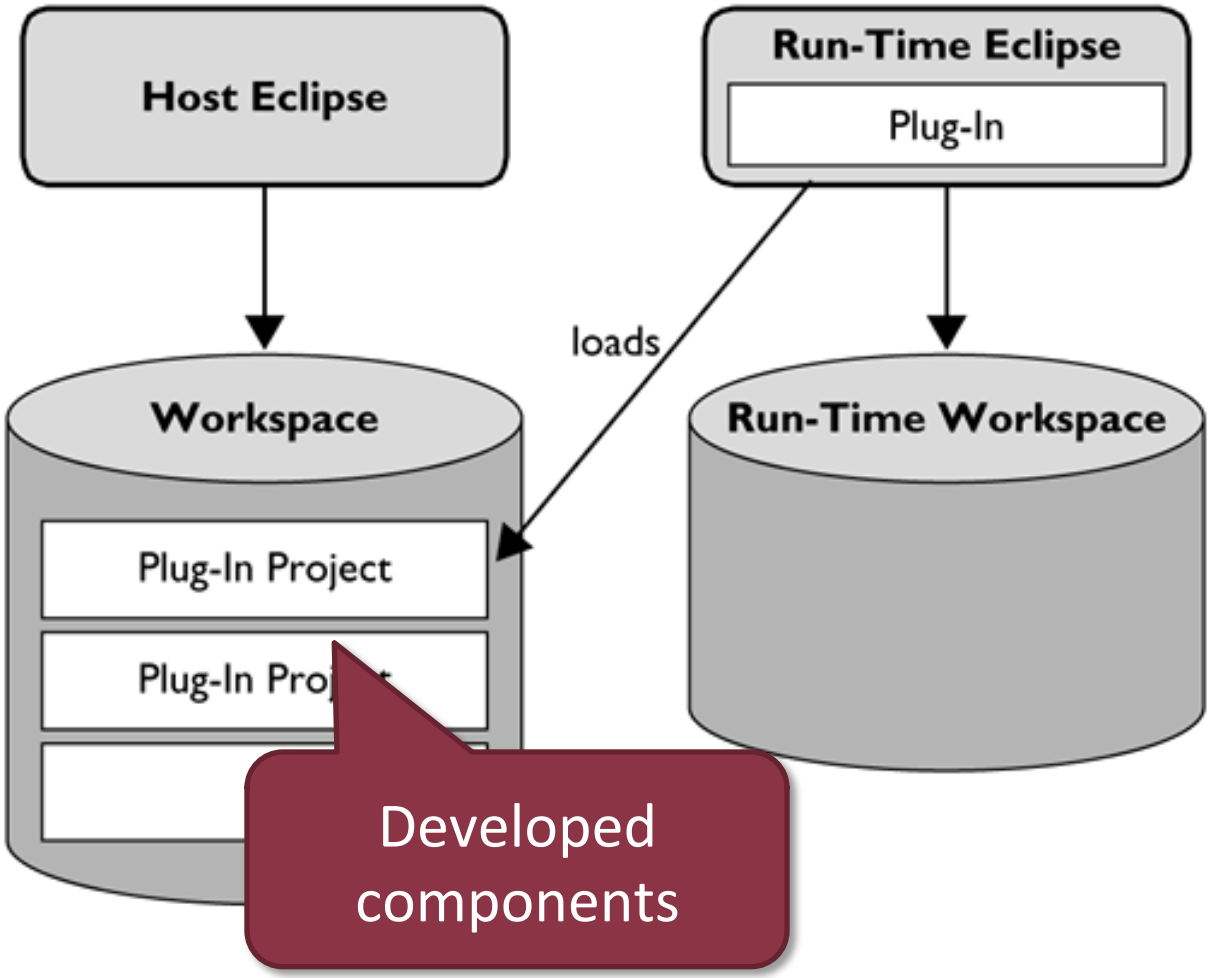
Execution


Host Eclipse
Run-Time Eclipse
Plug-In
loads
Workspace
Plug-In Project
Plug-In Project
...
Run-Time Workspace

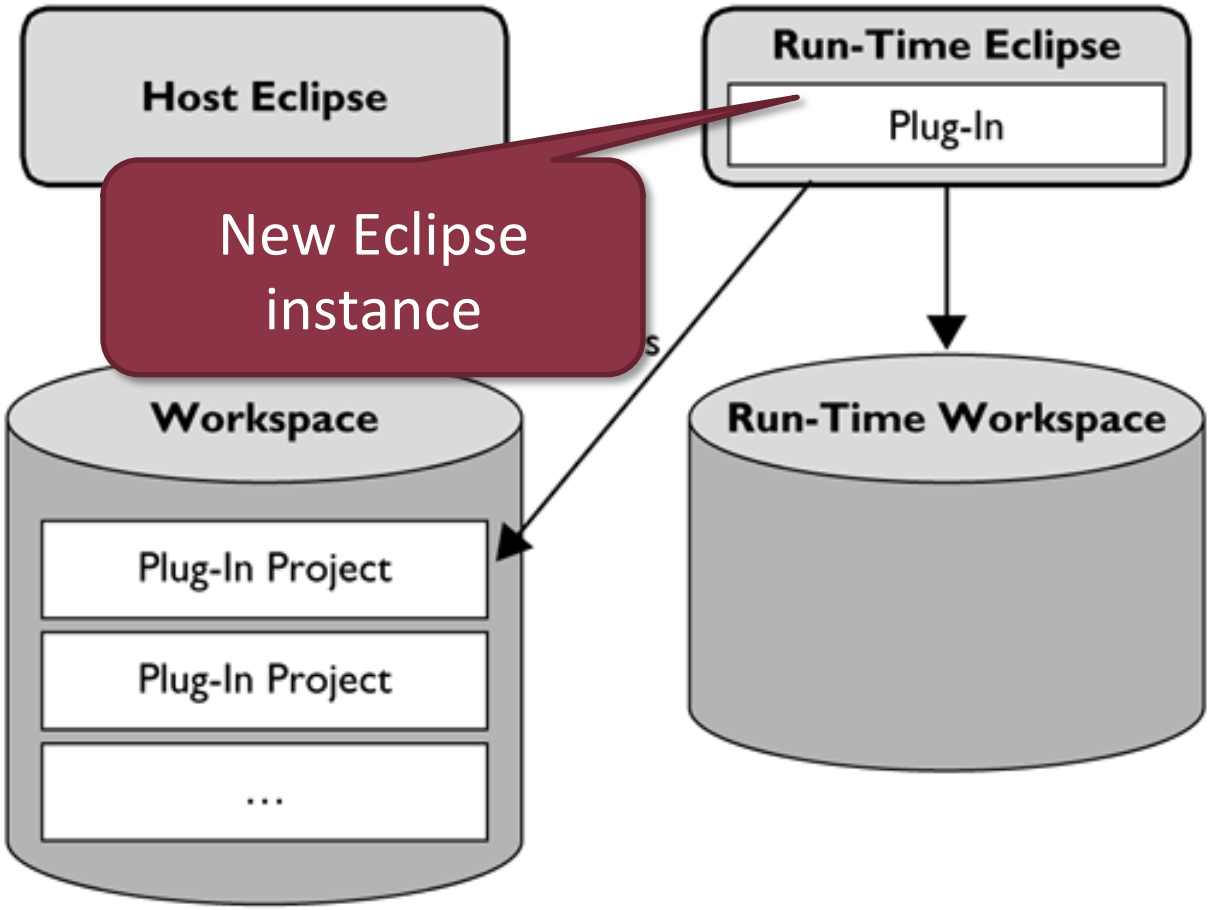
Execution


Work Environment
Host Eclipse
Run-Time Eclipse
Plug-In
loads
Workspace
Plug-In Project
Plug-In Project
...
Run-Time Workspace

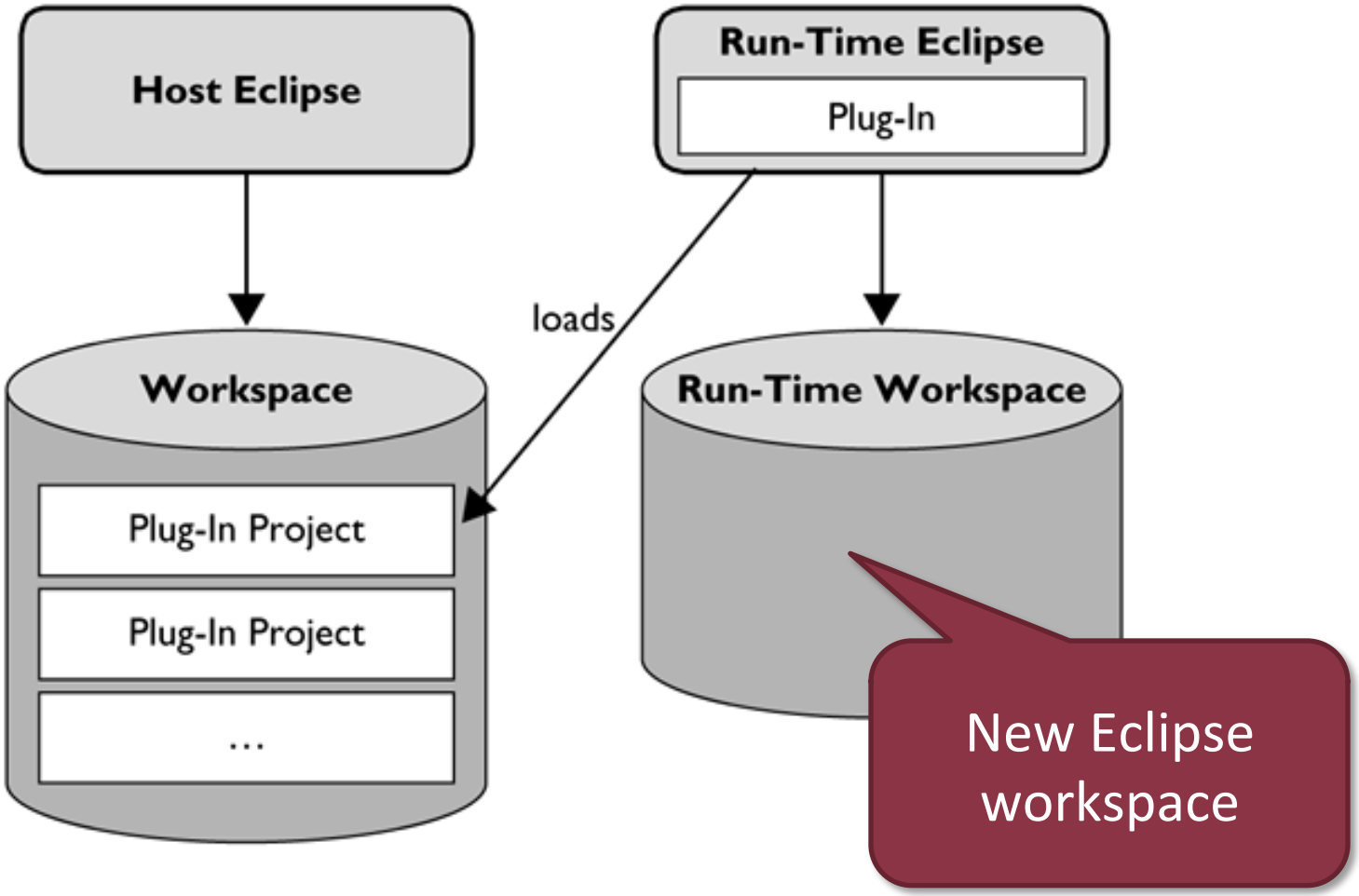
Execution


Host Eclipse
Run-Time Eclipse
Plug-In
loads
Workspace
Run-Time Workspace
Plug-In Project
Plug-In Pro
Developed components

Execution


Host Eclipse
Run-Time Eclipse
Plug-In
New Eclipse instance
Workspace
Plug-In Project
Plug-In Project
...
Run-Time Workspace

Execution


Host Eclipse
Run-Time Eclipse
Plug-In
loads
Workspace
Plug-In Project
Plug-In Project
...
Run-Time Workspace
New Eclipse workspace

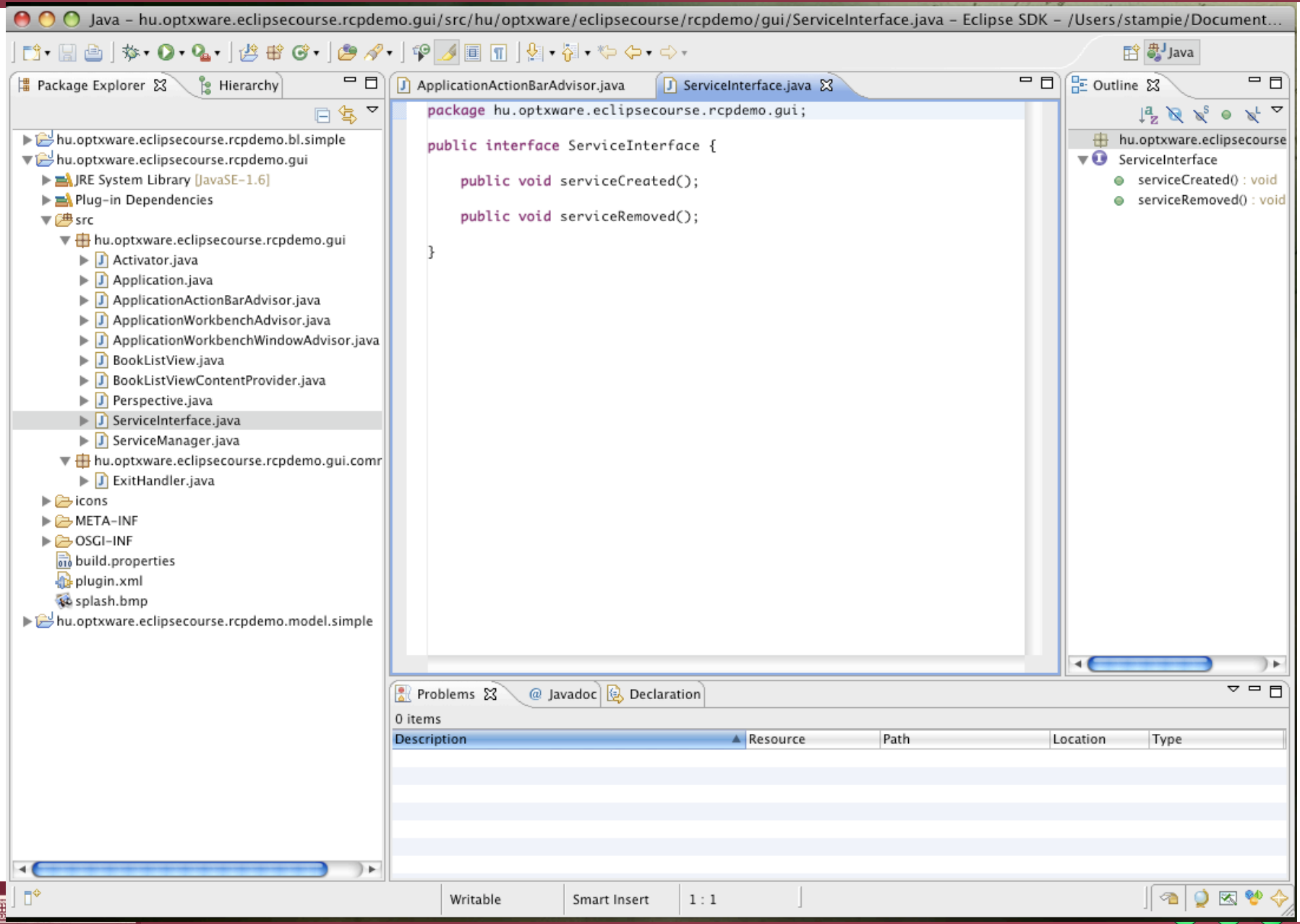
Common Extensions

Eclipse UI (3.x)

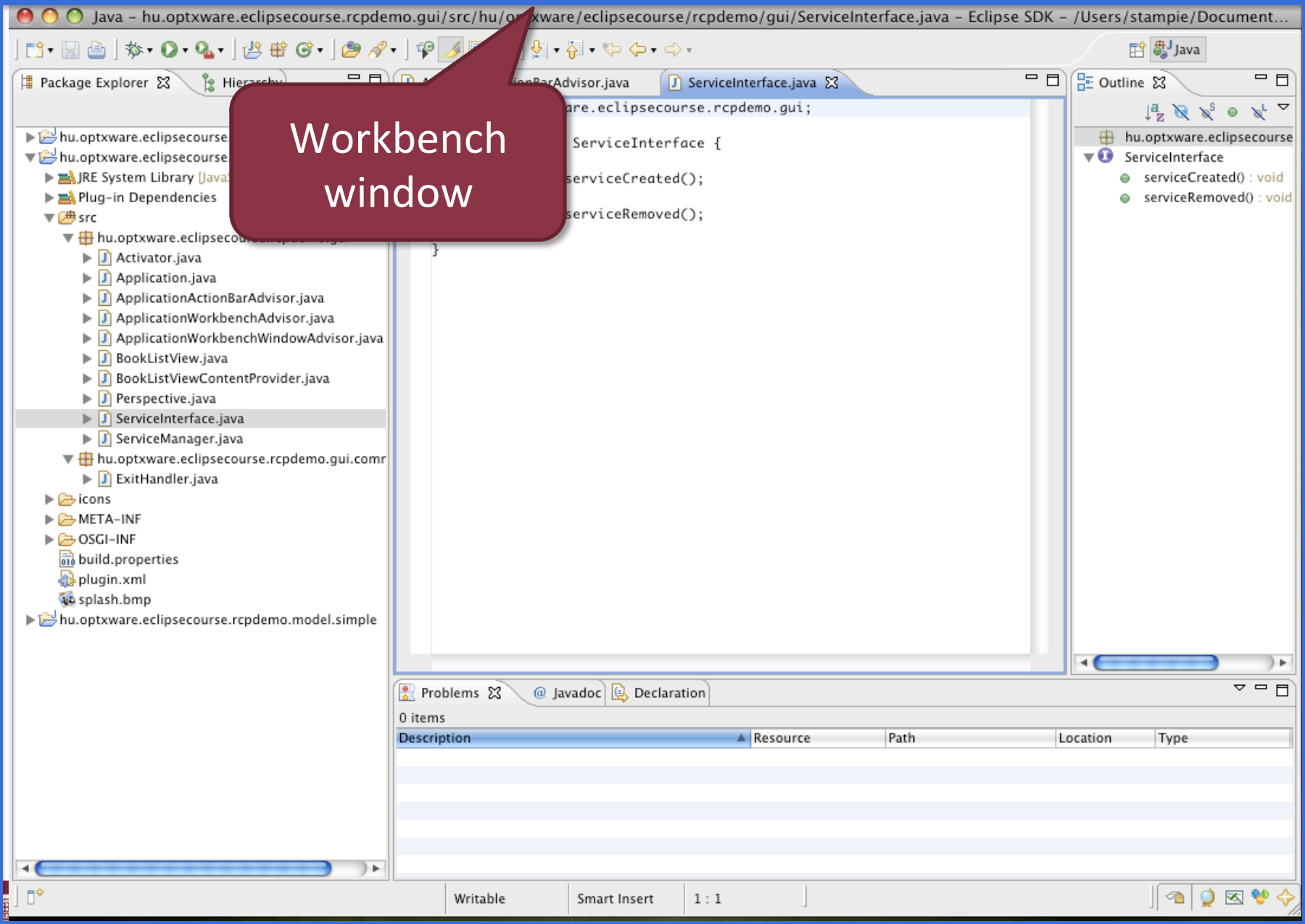
Workbench

- Workbench
  - A Window of an Eclipse-based application
  - Fixed set of elements supported
  - Configurable

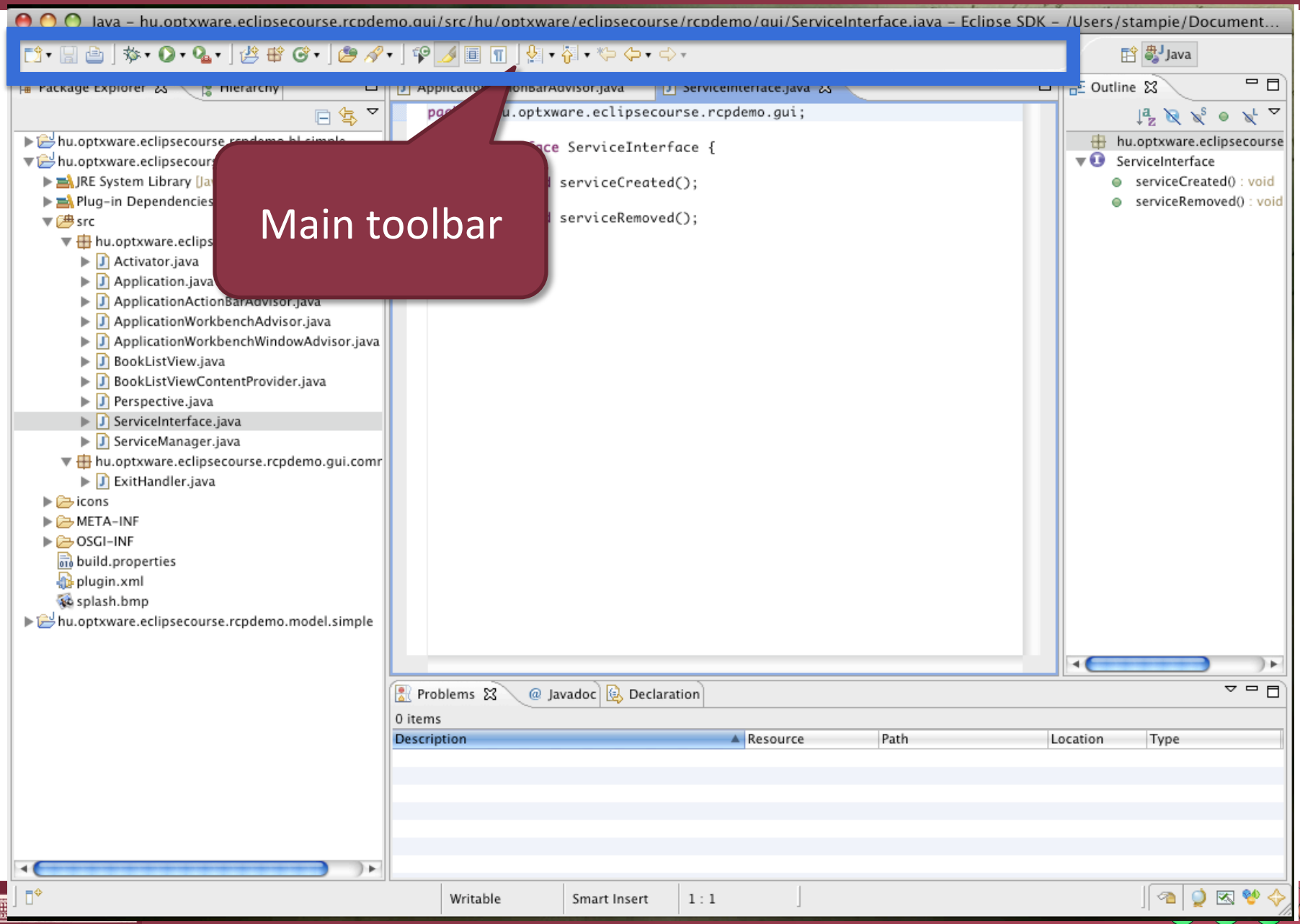
Workbench structure



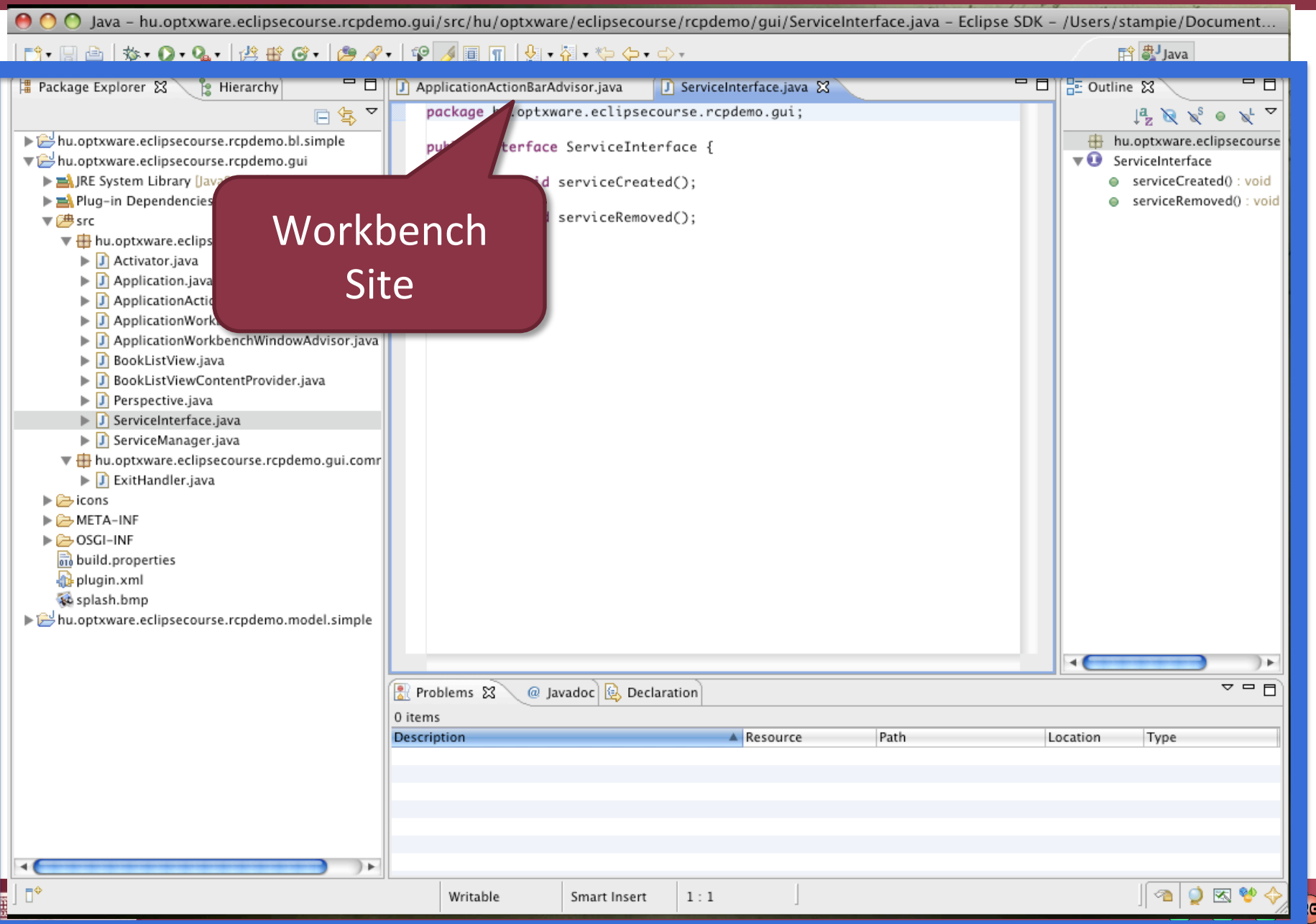
Workbench structure



Workbench structure



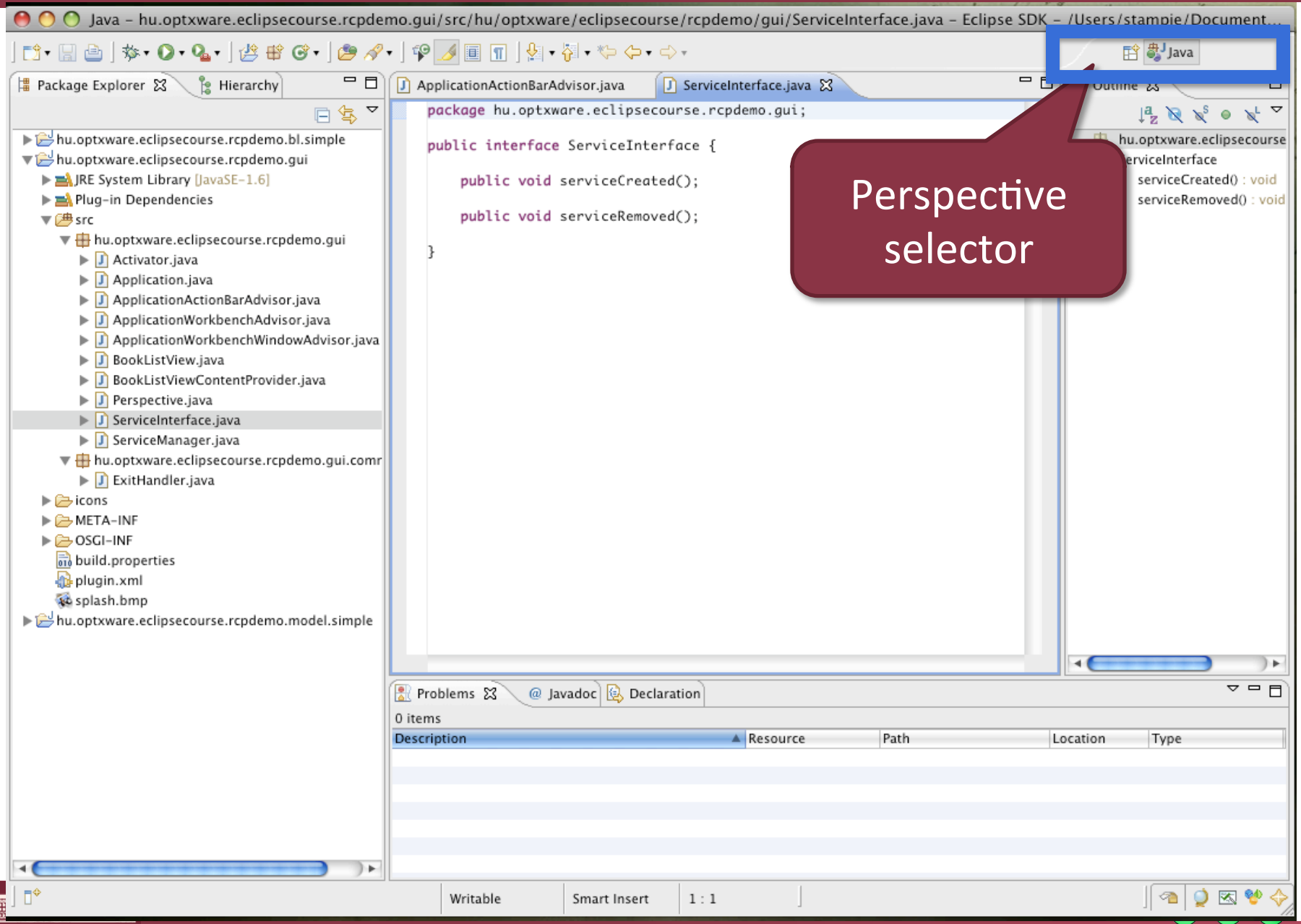
Workbench structure



Workbench Site

- Contains *Parts*
  - Editors (editor part)
  - Views (view part)
- Views are arranged according to selected *Perspective*

Workbench structure



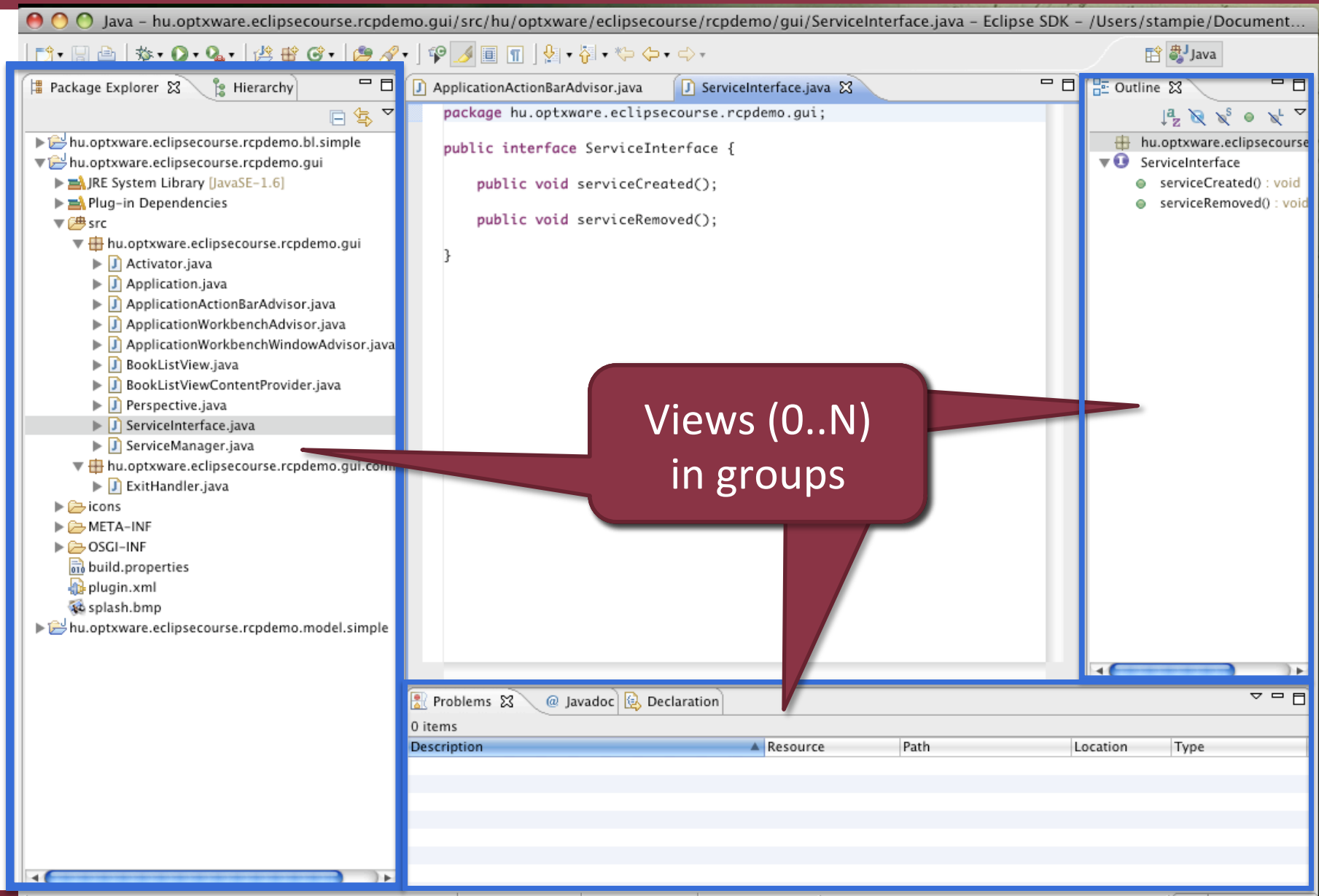
Perspectives

- A functional group of Views
  - Layout description as well
  - Task-specific collection
- Examples
  - Java Development
  - Plug-in Development
  - Debug
  - Team Synchronizing

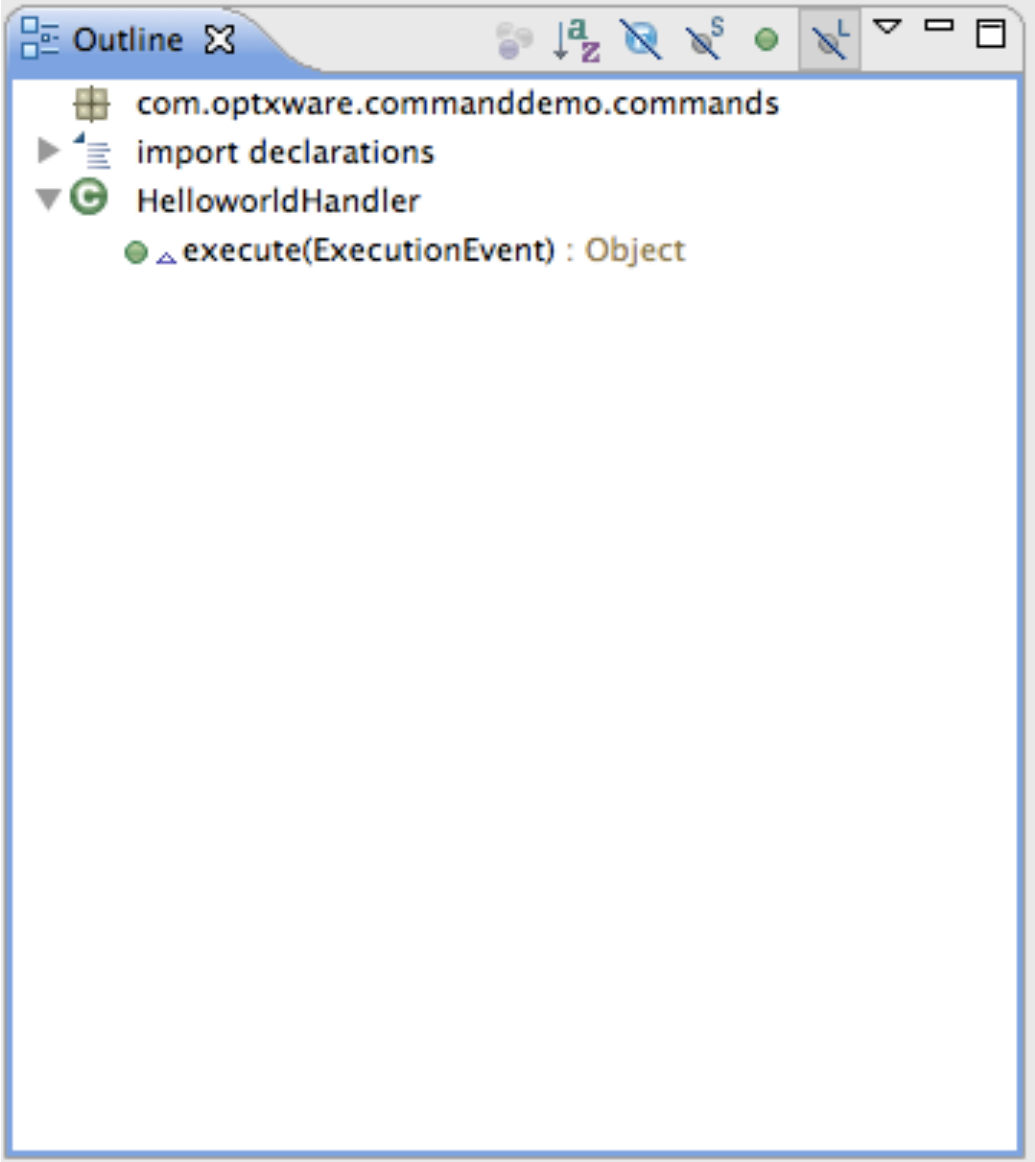
Perspective Implementation

- Perspective
  - Extension point: org.eclipse.ui.perspectives
  - Interface: IPerspectiveFactory
- Layouting support for views
  - Place all views related to editor area
- Further options
  - Perspective extensions can be used to extend an already defined view

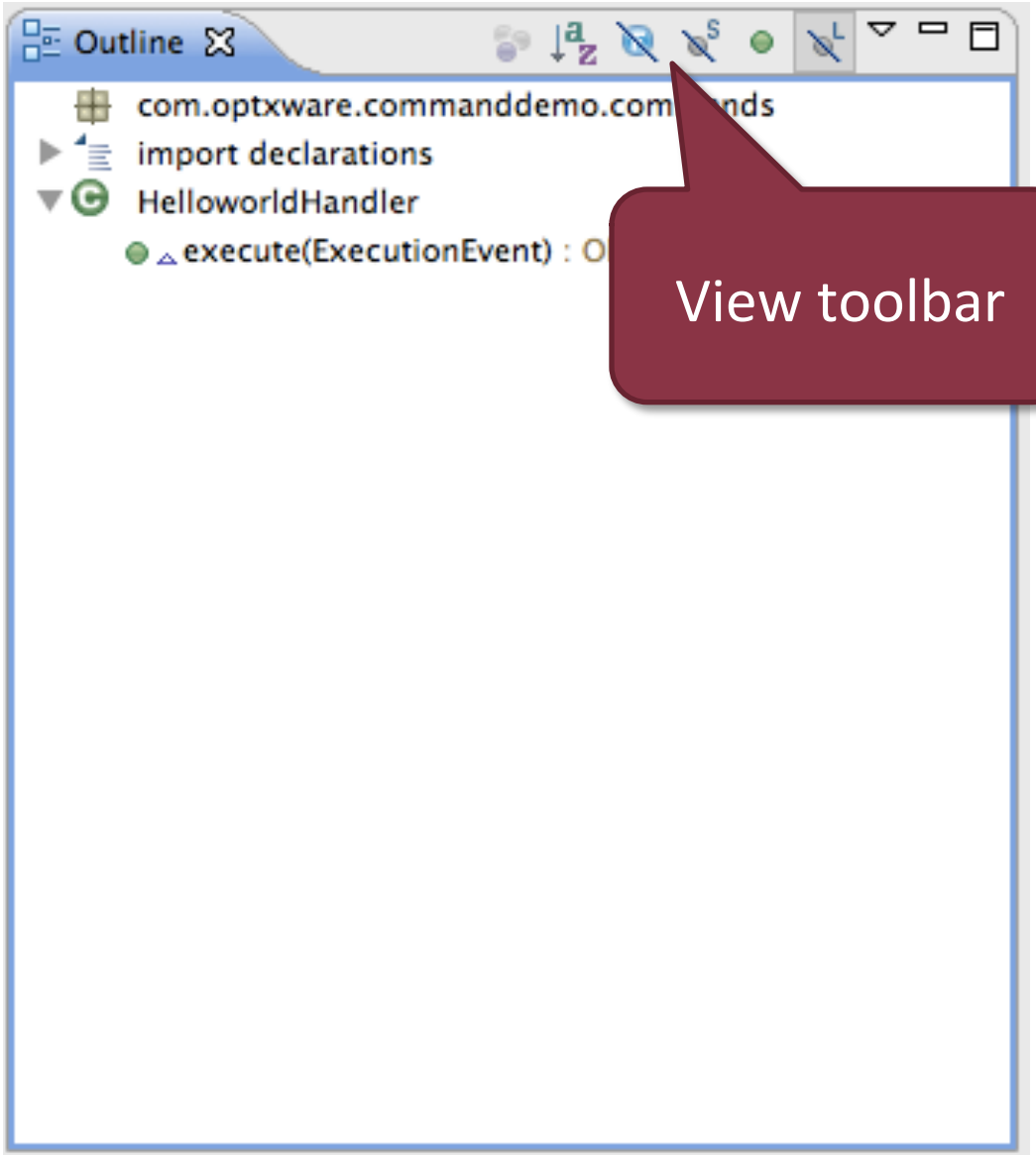
Workbench structure



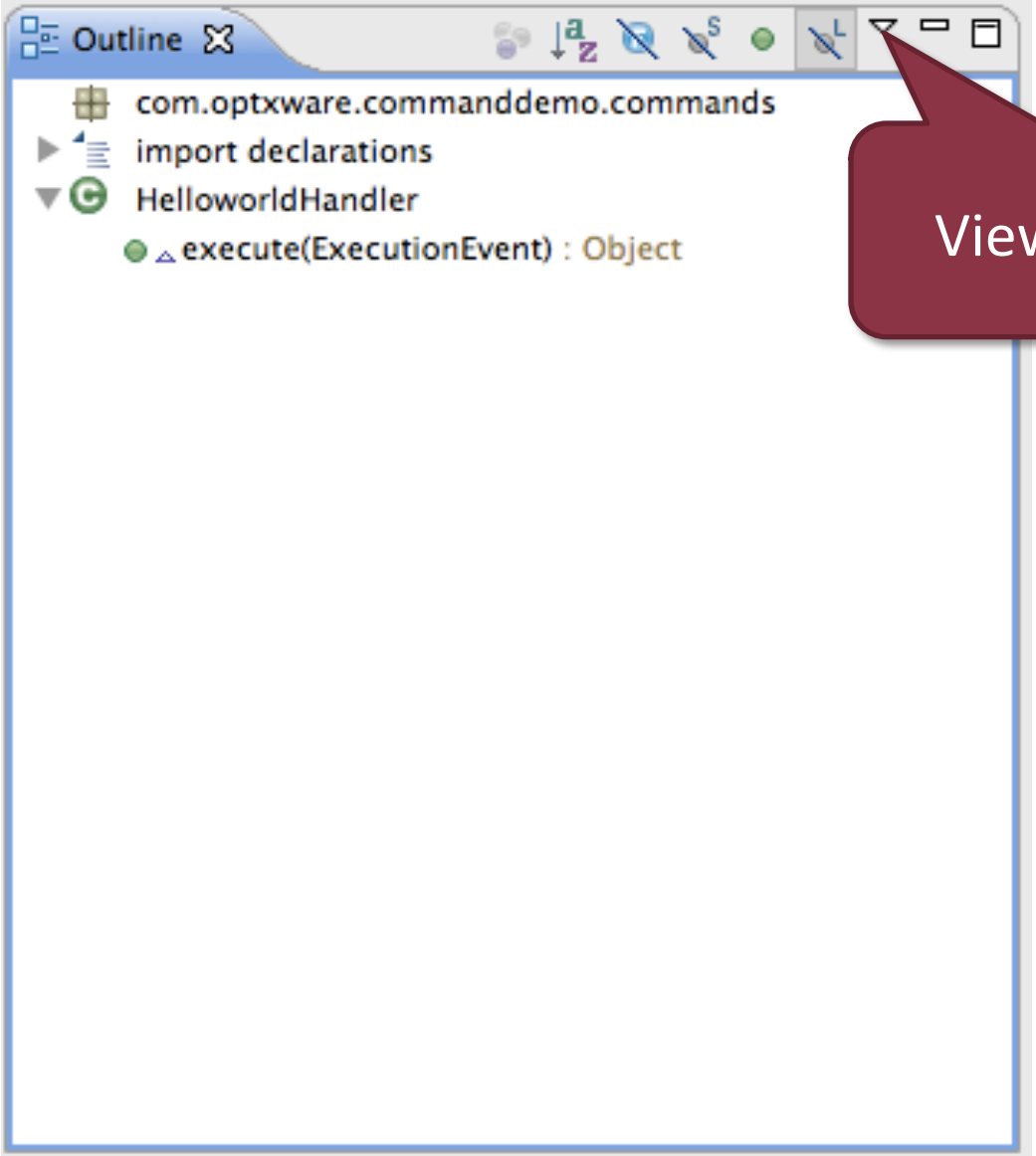
Structure of Views



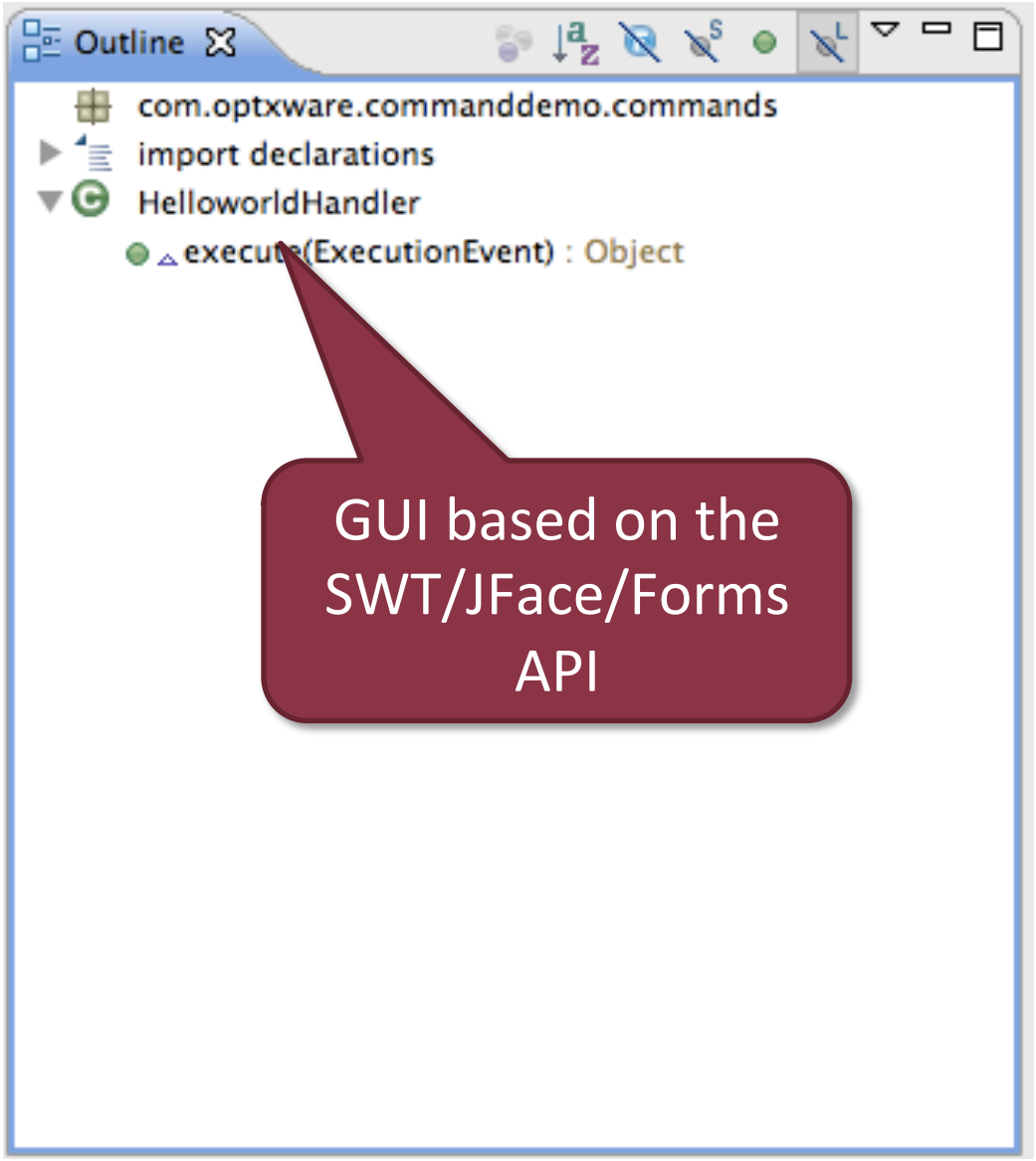
Structure of Views



Structure of Views



Structure of Views



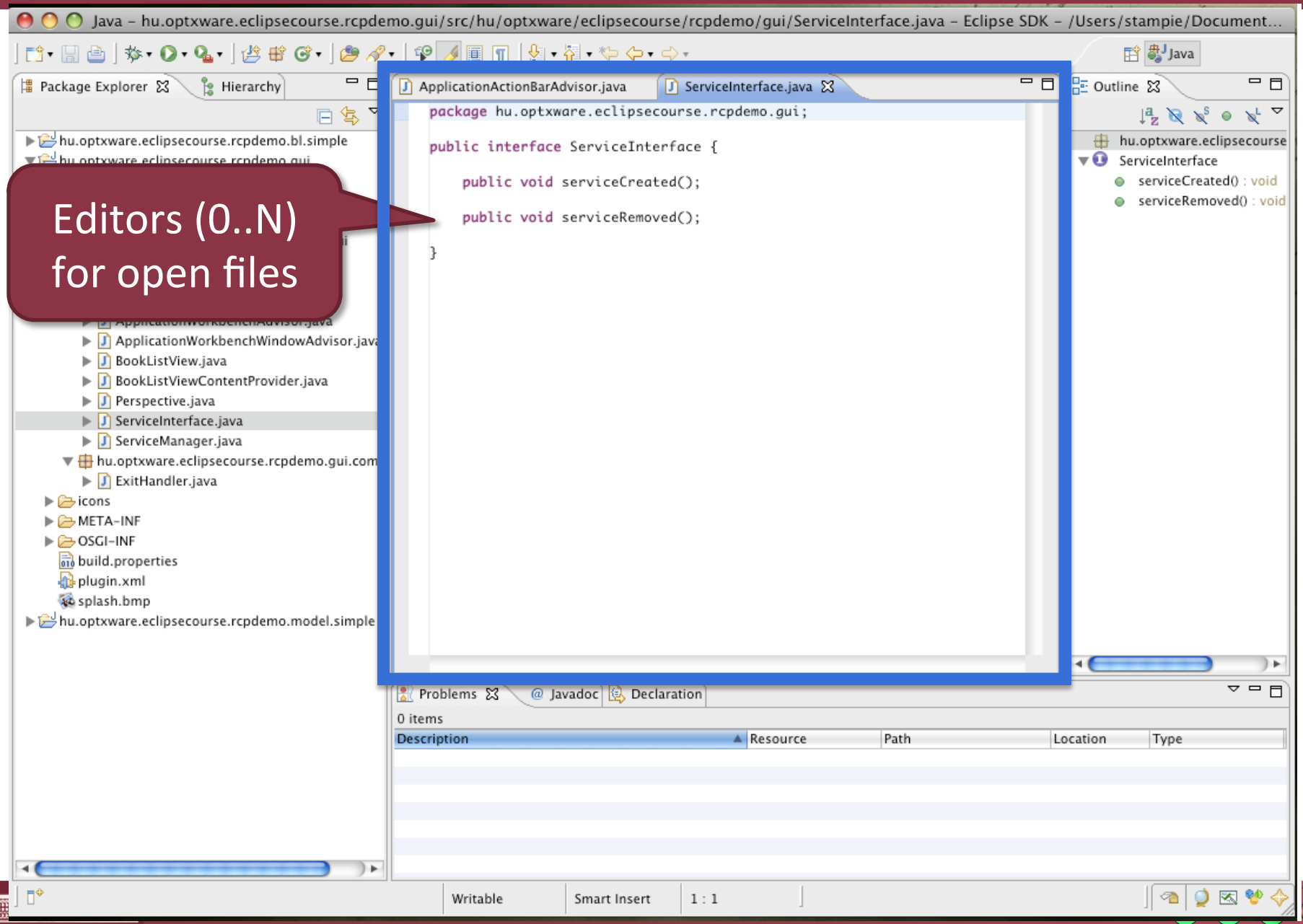
Views

- Helper elements for editors
  - Goal to display additional information
- May be global
  - Project Explorer
  - Error Log
- May depend on current editor/selection
  - Properties view – current selections properties
  - Outline view – quick overview of current editor contents

Implementing Views

- View
  - Extension point: org.eclipse.ui.views
  - Interface: IViewPart
- Goals
  - Presenting information
  - Editing information
  - Handling context changes
    - E.g. selection has changed

Workbench structure



Editor

- For editing purposes
  - Typically file editor
    - One of more files
  - But can be something else
    - contents of a database
    - compare editor
- Technologies
  - Textual (e.g. Java editor)
  - Form-based (e.g. plug-in manifest editor)
  - Graphical (e.g. UML editor)
  - Multi-page (each page may be different)

Editor registration

- Editor
  - Extension point: org.eclipse.ui.editor
  - Interface: IEditorPart
- Minimal functionality
  - Displaying editor
  - State display
  - Loading/saving data
- Building (compiling) is not the job of an editor!

Editor implementation

- createPartControls(Composite parent)
  - Defines the GUI elements of the editor
    - Can be any possible form
    - E.g. textual editors use a single multiline textedit component
- init(IEditorSite site, IEditorInput input)
  - Initializing editor
  - Loading data model
  - Presenting data model on widgets
  - Each editor must implement this

Editors – Managing Data Models

- IEditorInput: Data model handle
  - Minimal information that is required to load all contents
  - Must not contain the entire model
    - The handle remain open even after the editor closes
- Handle types
  - FileEditorInput
    - Describes a file in the workspace
  - CompareEditorInput
    - Compares and edits multiple version of file(s)
  - MultiEditorInput
    - For multi-page editors
  - Extensible with custom handle

Editors – Editor state

- `isDirty()`
  - Signals whether the contents have changed
  - If true, the *Save* command is enabled
- After editing the editor **must** send notification
  - `firePropertyChange(`*`PROP_DIRTY`*`);`
  - Pull notification: new value is not pushed
    - `isDirty()` will be called in the future

Editors – Saving the Data Model

- isSaveAsEnabled()
  - Describes whether a copy can be made to a new file
- doSave(IProgressMonitor monitor)
  - Handler for the system-wide *Save* command
  - Saving is to be coded manually
  - It is possible to show save progress
- doSaveAs()
  - Handler for the system-wide *Save as* command
  - Save dialog is handled manually
  - Saving is coded manually
  - Monitor is missing!

Editors – Additional concepts

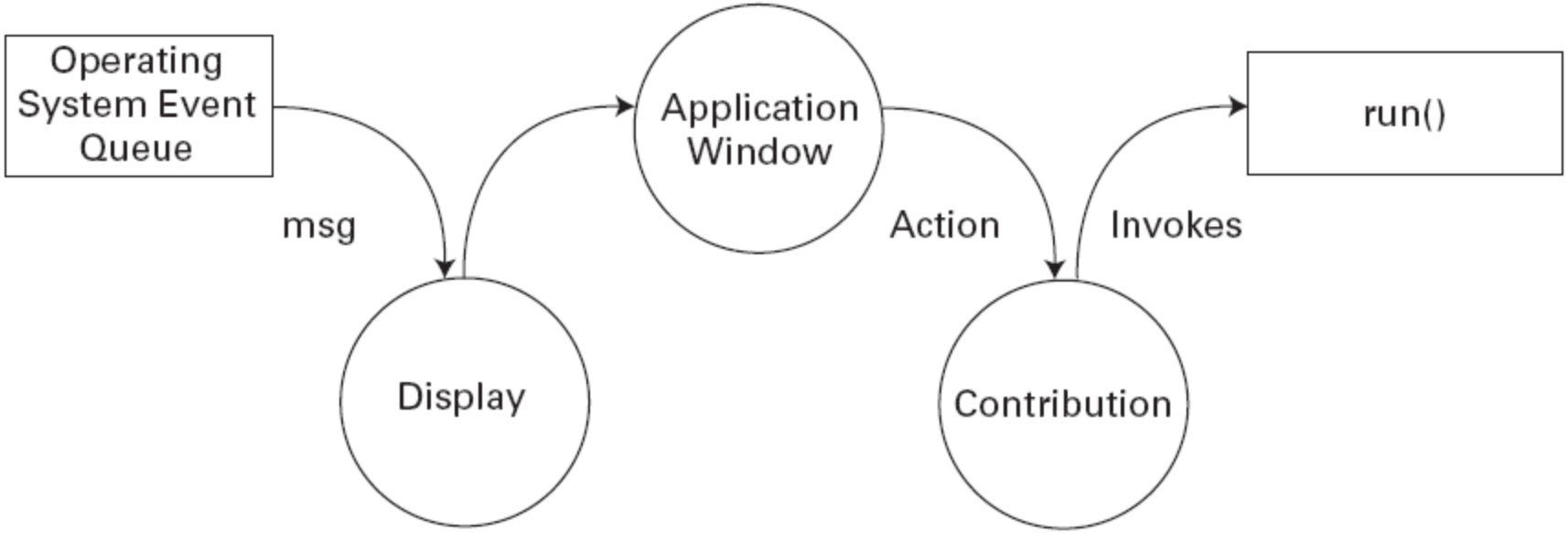
- Textual editor
  - AbstractTextEditor descendant
  - Xtext, EMFText or IMP projects
    - Higher level techniques
- Graphical editor
  - Graphical Editing Framework (GEF) project
  - Graphical Modeling Framework (GMF) project
- Multipage editors
  - MultiPageEditor descendant
  - Every page can be of a selected type

Command Framework

Actions vs Commands

- Menus, Pop-up menus and Toolbar items
  - Two approaches
    - JFace Actions
      - Older approach
      - Do not use in new projects (unless really necessary)
      - Just a very brief overview here
    - Command Framework (since Eclipse 3.3)
      - Eclipse 4.x API is based on this model

Actions



- Actions
  - Created by user interface events
  - For each action contributions are registered
    - The system chooses and executes the current contribution

Action registration

- Multiple extension points (depending of goal)
  - org.eclipse.ui.editorActions
    - For editor menus
    - Interfész: IEditorActionDelegate
  - org.eclipse.ui.viewActions
    - For view menus
    - Interfész: IViewActionDelegate
  - org.eclipse.ui.popupMenus
    - For pop-up menus
    - Interfész: IEditorActionDelegate v. IViewActionDelegate v. IObjectActionDelegate

Action implementations

- org.eclipse.jface.action.Action descendants
  - run()
    - Main execution method
  - Attributes
    - Presentation: text, toolTipText, image, etc.
    - Behavior: enabled, checked

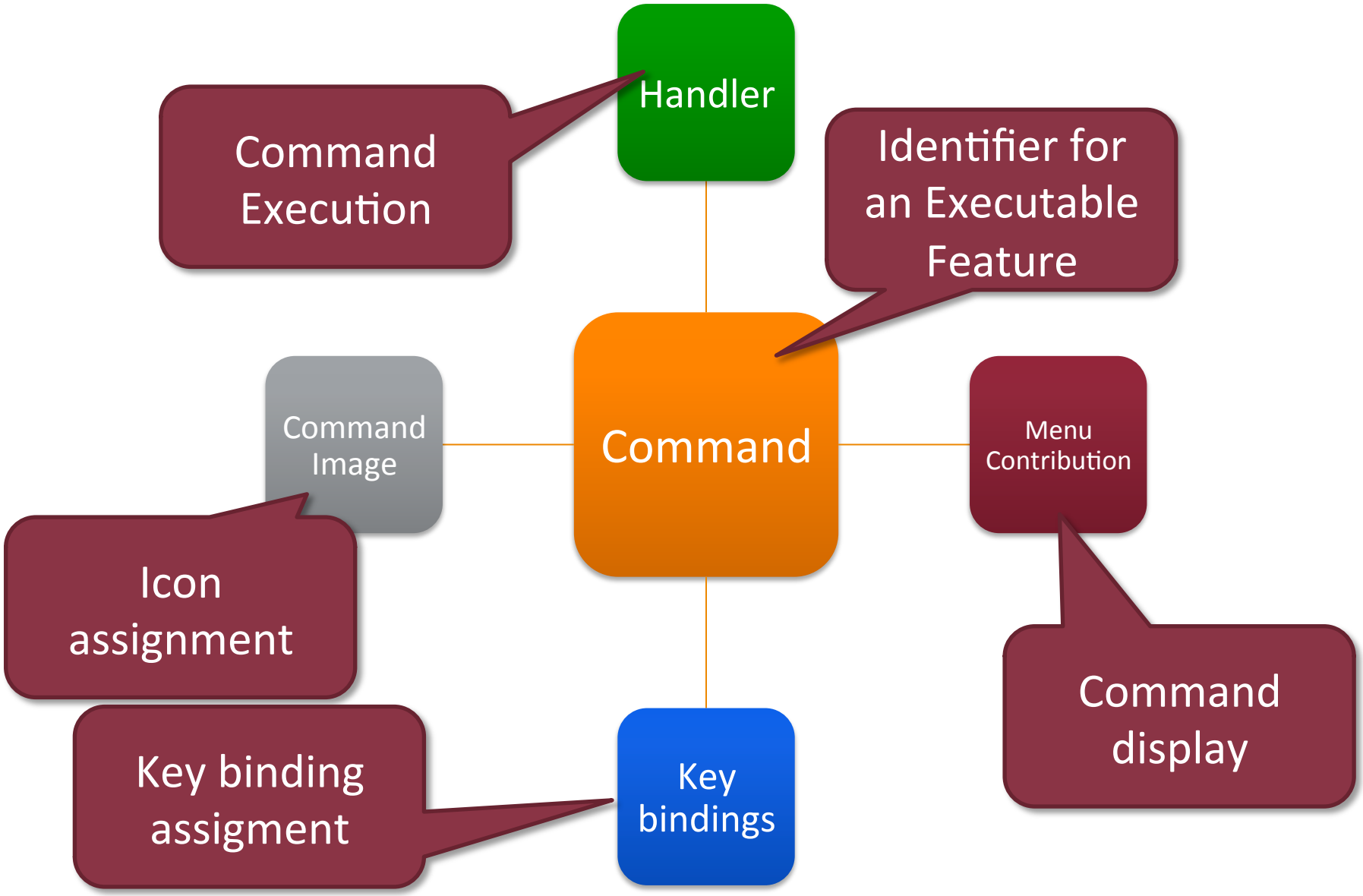
Use cases for item re-use

- Re-use supported
  - Existing command
    - Menu item/icon/binding
    - Command handler
  - What to re-use
    - Clipboard operations on view
    - Clipboard operation on editor
      - Textual
      - Graphical

Evaluation

- Multiple
  - Extension points
  - Java interfaces
- Presentation and behavior is not separated
- Problem
  - re-using Actions is problematic
    - XML duplication for registration
    - multiple Action interfaces are needed

Command Framework



Command

- Extension point: org.eclipse.ui.commands
- Mandatory attributes
  - Identifier
  - Name (for user interface)
- Optional attributes
  - Category
  - Default handler (not recommended to set)

Command example

```
<extension point="org.eclipse.ui.commands">
 <command
     description="Shows a Hello, world message.”
     id=”hu.mit.bme.commanddemo.helloworld"
     name="Display hello world">
 </command>
</extension>
```

Handler

- Extension point: org.eclipse.ui.handlers
- Mandatory parameters
  - Command identifier
  - IHandler or AbstractHandler descendant Java class
- Executable code
  - `execute(ExecutionEvent event)` method of Handler class
  - ExecutionEvent
    - Context information for execution
    - Use HandlerUtil for evaluation

Selecting the current Handler

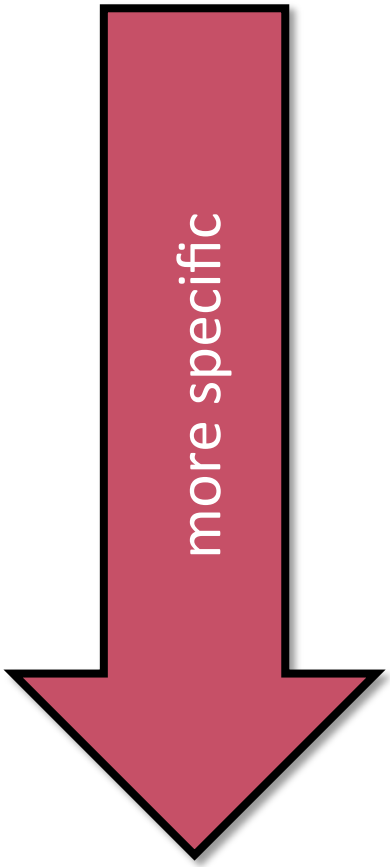
- Multiple handlers for a single command
- Choice required
  - activeWhen: activation condition (Core Expression - later)
  - enabledWhen: enablement condition (Core Expression)
- Choice algorithm:
  - All handlers with true activeWhen
  - Choose most specific condition (later)
  - If more than one handler remain, consider command disabled!

Core Expression

- Context condition setting in extensions
  - For evaluation plug-in code needs not to be loaded
- Possible conditions
  - Class checking
  - Count conditions
  - Comparisons
  - Logical operations (and, or, ...)
  - …

Conditions on

- Active Context
- Active Action Sets
- Active Shell
- Active Workbench Window
- Active editor
- Active part
- Current selection



Example: Core Expression

- The current selection contains an IFile

```
<with variable = “selection”>
  <iterate operator =”or”>
    <instanceof
        value =
  ”org.eclipse.core.resources.IFile”>
    </instanceof>
  </iterate>
</with>
```

Example: Core Expression

- The Java editor of JDT is active

```
<with variable = “activeEditorId”>
  <equals   value =
  ”org.eclipse.jdt.ui.CompilationUnitEditor”>
  </equals>
</with>
```

Example: Handler registration

- Handler active when Java editor is active

```
<extension point="org.eclipse.ui.handlers">
  <handler
    class=
  ”hu.bme.mit.commanddemo.handlers.helloworld"
    commandId=”hu.bme.mit.commanddemo.helloworld”>
    <activeWhen><with variable = “activeEditorId”>
      <equals

  value=”org.eclipse.jdt.ui.CompilationUnitEditor”>
      </equals>
    </with></activeWhen>
  </handler>
</extension>
```

Contribution

- Extension point: org.eclipse.ui.menus
- Describes the place of contribution
  - URL: "[Scheme]:[ID]?[Query]"
    - Scheme
      - Contribution type
      - Supported values: menu, popup, toolbar
    - ID
      - identifying the view/editor/toolbar
    - Query
      - more precise positioning inside the menu/toolbar
      - e.g. after the *Run* contribution

Contribution

- Attributes
  - Mandatory
    - Command ID
  - Optional
    - Contribution ID (for query part of URL)
    - Label (if not set, the command name is used)
    - Icon (use CommandImages instead)
  - Visibility conditions with Core Expression
    - visibleWhen child

Example: Contribution

- Toolbar icon for the main toolbar (technically a toolbar of toolbars)

```
<extension point="org.eclipse.ui.menus">
   <menuContribution     locationURI =
  "toolbar:org.eclipse.ui.main.toolbar">
    <toolbar id=”hu.bme.mit.commanddemo.toolbar1">
      <command commandId=
  ”hu.bme.mit.commanddemo.helloworld"
    id=”hu.bme.mit.commanddemo.helloworldInToolbar"
               label="Hello, world!"
               style="push"
               tooltip="Displays a hello, world
  dialog.">
          </command>
      </toolbar>
   </menuContribution>
</extension>
```

Example: Contribution

- Toolbar icon for the main toolbar … ar of toolbars)

URL

```
<extension point="org.ecl…s">
   <menuContribution      …I =
  "toolbar:org.eclipse.ui.m…in.toolbar">
     <toolbar id="hu.bme.mit.commanddemo.toolbar1">
       <command commandId=
  "hu.bme.mit.commanddemo.helloworld"
     id="hu.bme.mit.commanddemo.helloworldInToolbar"
                 label="Hello, world!"
                 style="push"
                 tooltip="Displays a hello, world
  dialog.">
          </command>
       </toolbar>
   </menuContribution>
</extension>
```

Example: Contribution

- Toolbar icon for the main toolbar (technically a toolbar of toolbars)

```
<extension point="org.eclipse.ui.menus">
   <menuContribution     locationURI =
  "toolbar:org.eclipse.ui.main.toolbar">
    <toolbar id=”hu.bme.mit.commanddemo.toolbar1">
      <command commandId=
  ”hu.bme.mit.commanddemo.hel
    id=”hu.bme.mit.commanddemo                    bar"
              label="Hello, wo
              style="push"
              tooltip="Displays a hello, world
  dialog.">
        </command>
      </toolbar>
   </menuContribution>
</extension>
```

New toolbar (needed on main toolbar)

Example: Contribution

- Toolbar icon for the main toolbar (technically a toolbar of toolbars)

```
<extens                    g.eclipse.ui.menus">
    <men                        locationURI =
   "tool                   se.ui.main.toolbar">
      <tool                id="hu.bme.mit.commanddemo.toolbar1">
        <command commandId=
   "hu.bme.mit.commanddemo.helloworld"
      id="hu.bme.mit.commanddemo.helloworldInToolbar"
                  label="Hello, world!"
                  style="push"
                  tooltip="Displays a hello, world
   dialog.">
            </command>
        </toolbar>
    </menuContribution>
</extension>
```

Command ID

Command Icons

- Extension points: org.eclipse.ui.commandImages
- Mandatory attributes
  - Command ID
  - Icon
- Goal:
  - Assign icons to command
- Contribution should not contain the default icon
  - Multiple contributions of the same command!
  - Contribution might reside in a different plug-in

Example: Command Icons

```
<image

  commandId=”hu.bme.mit.commanddemo.helloworld”
   icon=”icons/sample.gif”>
</image>
```

Key bindings

- Extension point: org.eclipse.ui.bindings
- Goal: assigning key bindings to commands
- Mandatory attributes:
  - Sequence: the key sequence
    - Emacs style longer sequences also supported
  - SchemeID: key binding scheme selection
  - ContextID: selecting where the binding is active
  - CommandID
- Optional attributes:
  - Platform: for platform-specific bindings
  - Locale: for language-specific bindings

Example: binding

```
<key
 sequence =
  "M2+F5”
 commandId =
  ”hu.bme.mit.rcpdemo.list”
 schemeId =
  "org.eclipse.ui.defaultAcceleratorCo
  nfiguration”
 contextId =
  "org.eclipse.ui.contexts.dialog" />
```

M2: modifier key; platform-dependent value

Further Sources

- Eclipse Commands Tutorial
  - http://www.vogella.de/articles/EclipseCommands/article.html
- Eclipse Platform Architecture
  - http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/guide/arch.htm