Introducing Eclipse

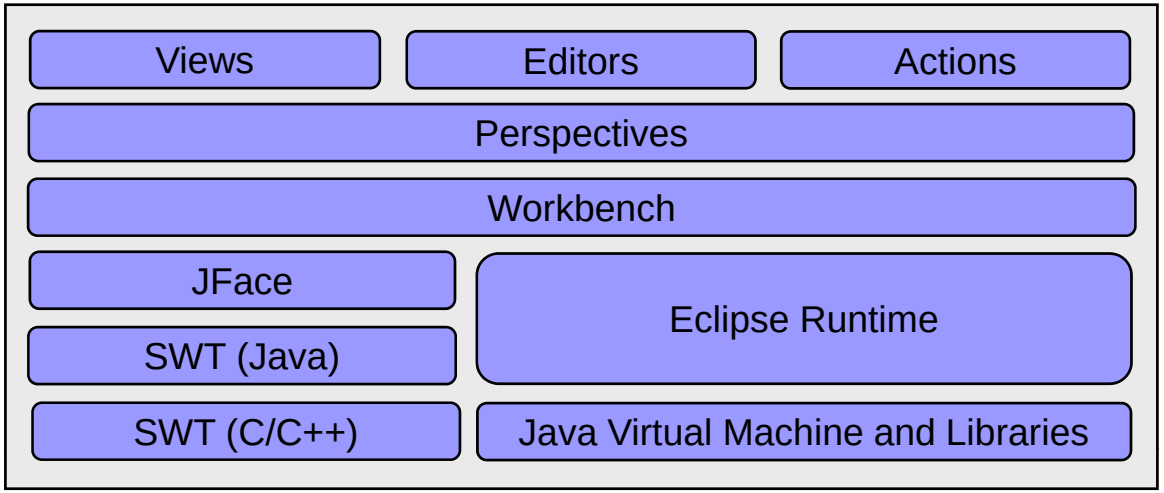
Plug-ins, RCP and SWT

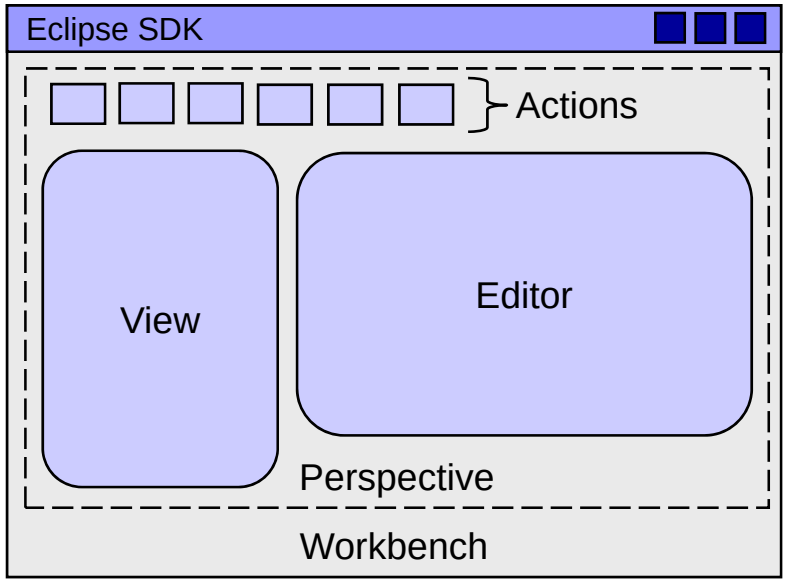
Building On Top Of Eclipse

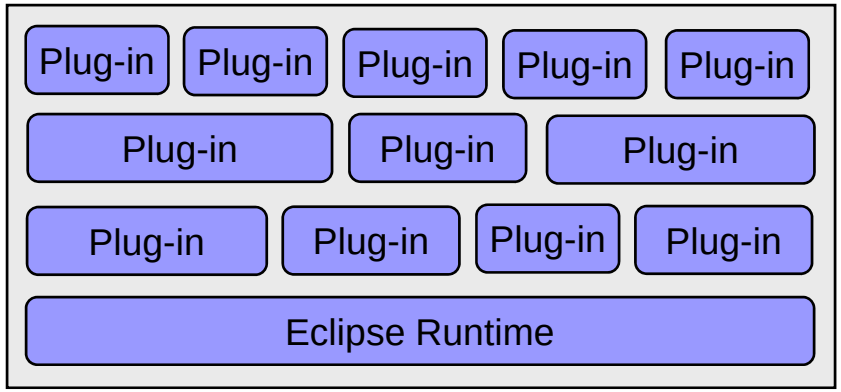
- A majority of Java projects use Eclipse as their IDE
  - However there is a big difference between *developing with* Eclipse and *developing on top of* Eclipse
- From the beginning Eclipse was explicitly designed as a fully organic, contribution driven project
  - Apart from a small kernel all of Eclipse consists of plug-ins
  - This is in keeping with the Smalltalk philosophy, where the objects that make up any system can be viewed and extended
    - Intentionally blurring the distinction between user and developer
- Other IDE's also support extensions
  - But these are contributions added on top of a monolithic base

Individual Eclipse Technologies

- IBM created their own GUI library for Eclipse
  - This is the Standard Widget Toolkit (SWT)
  - It provides containers, widgets, events and layouts
- JFace is a set of high level controls based on SWT
  - Such as dialogs, wizards and source code viewers
- The Workbench is the most fundamental GUI container which all visual components plug into
  - There is a lot of overlap between the Workbench and JFace
- Perspectives organize resources to accomplish tasks
  - Such as editing code, debugging programs and drawing pictures
  - Perspectives contain editors, views and actions (menus etc…)







Types of Eclipse Contribution

- There are many ways of contributing to Eclipse
  - Depending on the type of task you are trying to accomplish
- Additions to the Java IDE can be simple plug-ins
  - They will only contribute additional actions to menus and views
- Integrating new tools requires more complex plug-ins
  - These add new views to the standard perspectives
  - They should define extension points that allow other developers to extend your contribution via their own plug-ins
- Rich Client Platform applications are the most complex
  - They contribute a completely new perspective to the workbench
  - This is a novel but increasingly important use of Eclipse

The Standard Widget Toolkit

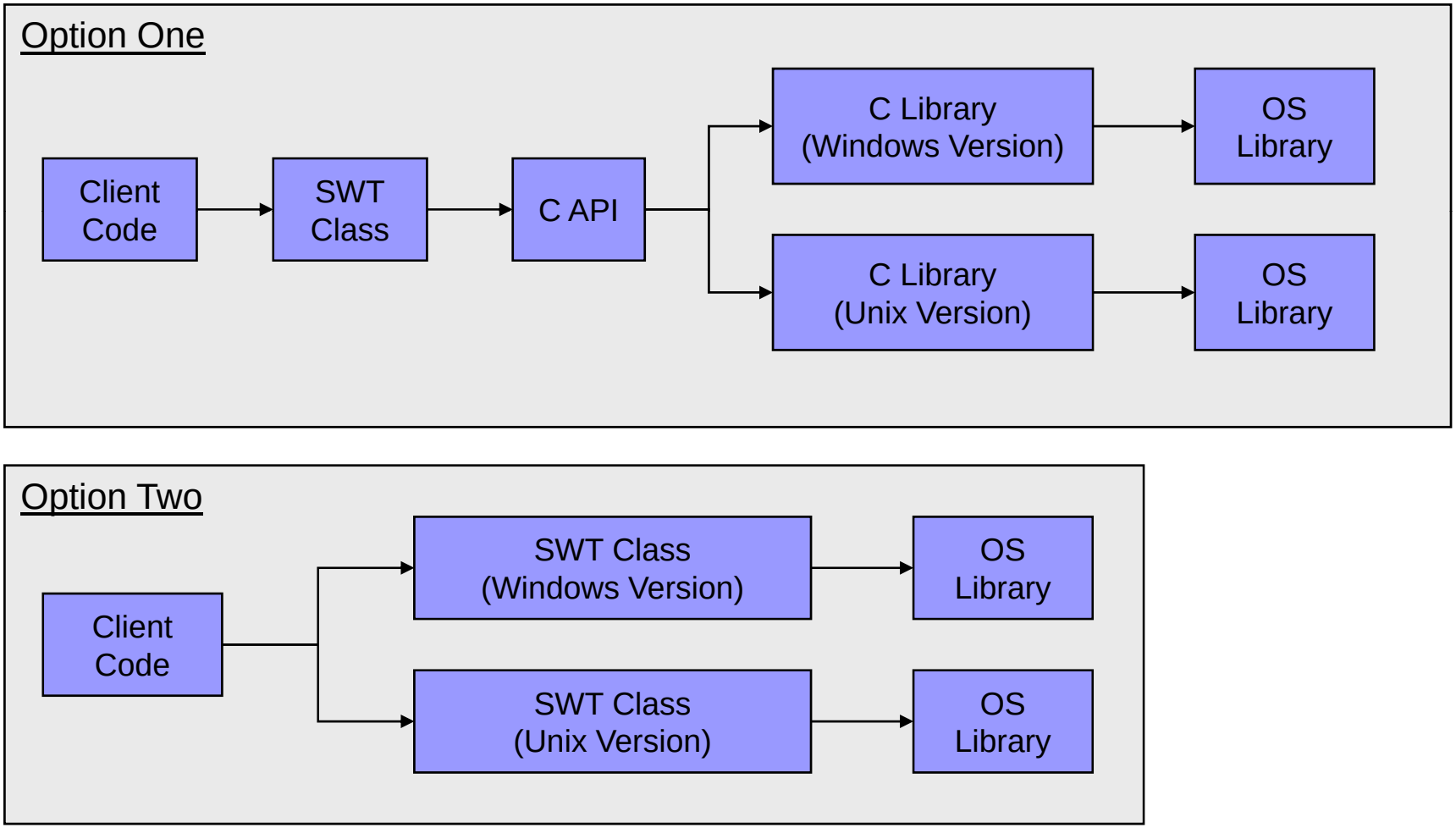
Writing GUI's in SWT

Sample Content

Introducing SWT

- Swing and SWT are mirror images
  - Swing widgets are mostly rendered by Java code
    - With a few instances where native controls are used
  - SWT widgets map directly to native controls
    - With a few additional widgets rendered in Java
- IBM had two motivations for creating SWT
  - Dissatisfaction with the speed and features of AWT/Swing
  - Extensive in-house experience developing cross-platform GUI libraries as part of Smalltalk and VisualAge
- Both libraries have continued to evolve over time
  - Swing is now faster while SWT been ported to more platforms
  - The popularity of Eclipse has balanced Swings official status

Which Design Does the SWT Use?



Which Design Does the SWT Use?

- SWT uses option two from the previous slide
  - There is a separate set of Java classes for each platform
    - E.g. all SWT 'Button' classes have the same public members but completely distinct and independent implementations
  - Each class contains a direct call to a native method
    - The calls are made through the Java Native Interface (JNI)
    - The core SWT widgets contain no extra C implementation
- A platform neutral C API would allow one Java library
  - But such an API would be very hard to design and maintain
  - Also it would be harder to debug the Java/OS interface

Benefits and Drawbacks of SWT

- SWT has two main advantages
  - Speed because rendering is done in optimized native code
  - Complete fidelity between the appearance of the GUI and the current platforms look and feel (because both are the same)
- The SWT approach has drawbacks
  - An SWT port has to be available for the platform
  - The need to validate the GUI on each platform
  - The need to install native libraries onto the client
  - The programming model is more complicated
    - See the following slides for more details…

A Simple SWT Application

```
public class HelloSWT {
	public static void main(String[] args) throws IOException {
		Display display = new Display();
		Shell mainWindow = new Shell(display);
		mainWindow.setText("Hello SWT");
		mainWindow.setLayout(new FillLayout());
		final String imageLocation = PATH_TO_IMAGE_FILE;
		ImageData data = new ImageData(new FileInputStream(imageLocation));
		Image image = new Image(display,data);
		Button button = new Button(mainWindow,SWT.PUSH);
		button.setImage(image);
		mainWindow.pack();
		mainWindow.open();
		while(!mainWindow.isDisposed()) {
			if(!display.readAndDispatch()) {
				display.sleep();
			}
		}
		display.dispose();
	}
}
```

The Display Class

- Every SWT interface needs a Display object
  - Which manages the interaction between SWT and the OS
  - The Display is disposed at the end of the program
    - This disposes of all the widgets within the GUI
    - Widgets in SWT must be explicitly released
      - To release the OS resources they have a hold of
- SWT requires that you create an explicit event loop
  - 'Display.readAndDispatch' tries to retrieve and handle the next event from the queue created by the operating system
    - It returns true if there was an event to dispatch and false otherwise
  - 'Display.sleep' causes the calling thread to sleep
    - Until an event becomes available on the OS queue

Comparing SWT and Swing Coding

- In SWT the convention is that widgets are passed their parent component in the constructor
  - Every widget has a parent, which cannot be changed
- Widgets are typically configured by passing a bitmap of constant values in the constructor
  - All the constants are defined within the ‘SWT’ class
  - E.g. ‘Group g = new Group(this,SWT.BORDER)’
  - E.g. ‘Text t = new Text(this,SWT.MULTI | SWT.BORDER)’
- Subclassing widgets is not encouraged
  - Because the implementation of widgets is platform specific there is a danger that your derived class wont work consistently

Comparing SWT and Swing Coding

- Widgets must be explicitly released
  - Because they have a lock on OS resources
  - Just like file handles and network connections
- The ‘dispose’ method is called to:
  - Hide the widget and all its children
  - Release all the associated resources
  - References in the parent are set to null in order to make the widget eligible for garbage collection
- Fortunately disposal works in cascades
  - Disposing a container disposes its children
  - Disposing a menu disposes any sub-menus

Listening for Events in SWT GUI's

- SWT has two kinds of event handling
  - Event handlers can be both typed and un-typed
- There is one generic event handler interface that can be used to handle all types of event
  - The 'Listener' interface defines a single method with the signature 'void handleEvent(Event event)'
  - Implementations of the interface are attached with a constant value that specifies the type of the event
    - E.g. 'button.addListener(SWT.MouseEnter,myListener)'
- Conventional typed event handlers also exist
  - E.g. 'button.addMouseListener(myMouseListener)'

Layout Managers in SWT

- As with AWT/Swing absolute coordinates cannot be used to place components
  - Instead a layout object is used to implement a platform-independent strategy for positioning widgets
  - Layouts aren't associated with any OS resources and therefore don’t need to be disposed
- Each widget can supply its preferred minimum size
  - Widgets that expand or scroll may need an explicit size
- Complex GUI’s are created by combining layouts
  - A ‘Shell’ can contain many ‘Group’ objects, each of which can contain multiple widgets using its own layout

Using Fill Layout and Row Layout

- ‘FillLayout’ is the simplest algorithm
  - It positions the widgets either horizontally or vertically with all the available space distributed evenly
    - This is rarely the behaviour you are looking for…
- A ‘RowLayout’ is slightly more complex
  - Widgets are sized as they are positioned
    - If the ‘pack’ field is set to true then each component is positioned with its preferred minimum size
      - Otherwise each control uses the dimensions of the largest
    - Any space left over is not used, including after resizing
  - You can explicitly set the size of individual widgets
    - By supplying an instance of the ‘RowData’ class

Using a Grid Layout

- A ‘GridLayout’ divides the available space into a grid
  - Its complexity is midway between the ‘GridLayout’ and much feared ‘GridBagLayout’ in Swing
- Only the number of columns is set explicitly
  - The number of rows is determined by the number of widgets added to the composite control
  - By default the columns are of different sizes
    - Each column is a wide as the widest widget inside it
    - You can set all columns to be as wide as the widest one
- ‘GridData’ objects can be used for advanced positioning
  - The object specifies how many cells in the grid a widget should fill and what should be done with any surplus space

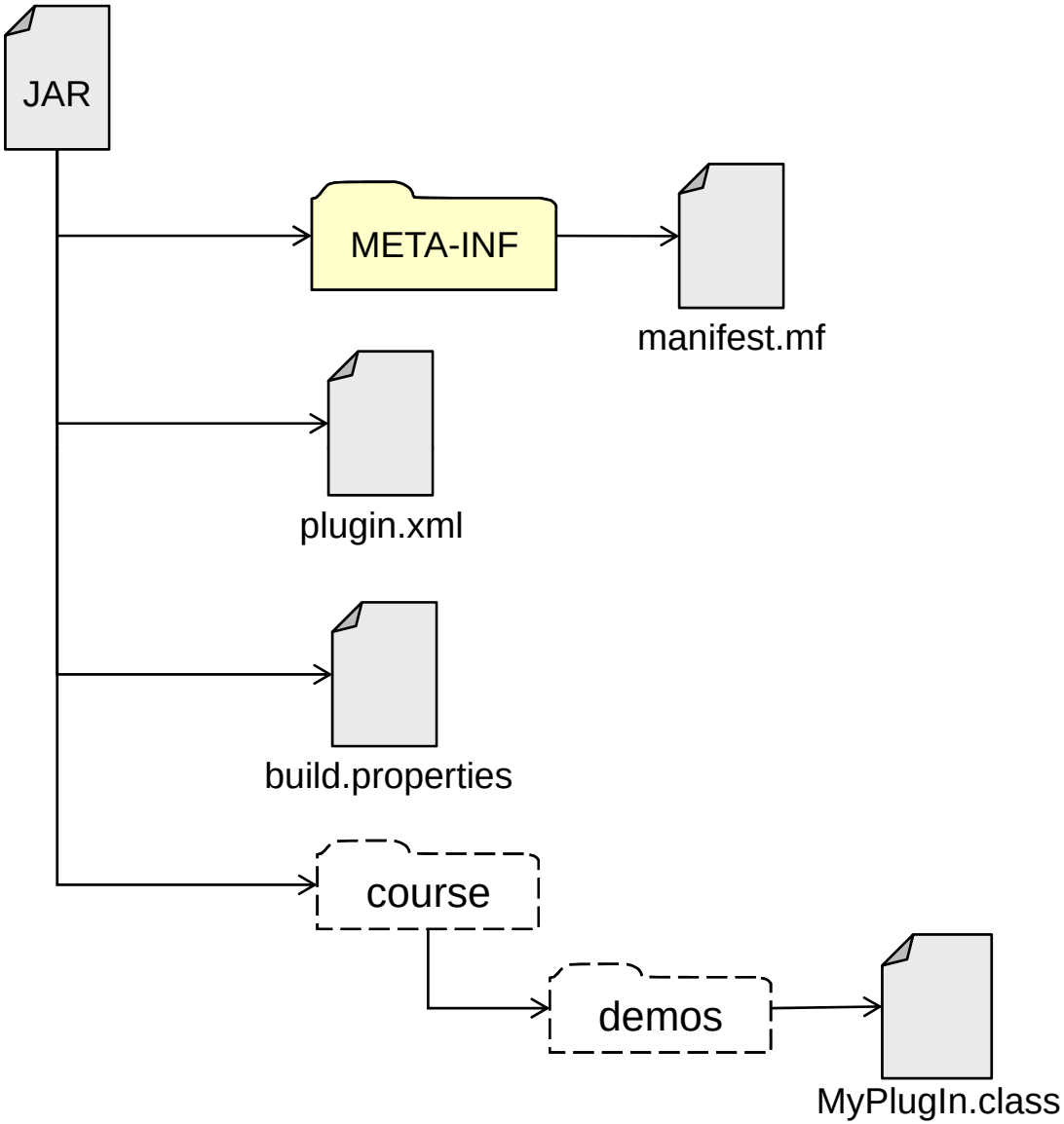
Using a Form Layout

- ‘FormLayout’ is the most complex manager
  - It positions widgets in relation to the edges of the parent container and of other widgets
  - The relationship can be expressed as an absolute value or as a percentage of screen size
- Each widget is added with a ‘FormData’ object
  - This specifies the height and width of the widget plus up to four ‘FormAttachment’ objects
    - Representing the top, bottom, left and right sides of the widget
  - An attachment defines where to position one edge of the widget
    - The most common use of this is to position one widget immediately below and to the right of another one
    - Attaching at opposing edges makes the widget stretch

Extending Eclipse

Writing Eclipse Plug-Ins

- The structure of a plug-in is simple
  - ‘plugin.xml’ is the deployment descriptor
    - It specifies the id of the plug-in, the points at which it extends Eclipse and its dependencies on other plug-ins
  - ‘build.properties’ defines names for magic numbers
    - These are used in ‘plugin.xml’ with a ‘%’ prefix
  - Class files are deployed as normal
    - Remember to ensure the package hierarchy is preserved
- The plug-in is deployed as a Java Archive
  - Which is then placed inside ‘ECLIPSE_HOME/plugins’
    - You can see it via ‘Help->About Eclipse SDK->Plug-In Details’



Developing and Testing Plug-Ins

- When you attempt to run a plug-in project Eclipse starts a new instance with the plug-in loaded
  - You can create and/or load projects into the new instance to verify your plug-in works
  - Any messages printed to ‘System.out’ appear in the console of the original copy of Eclipse
- Testing can be automated using PDE JUnit
  - This launches a new Eclipse workspace and runs all the test methods for your plug-in inside it
  - Typically in the ‘setUp’ method you create a new project for testing and populate it with types
    - Utility classes are available to simplify this

```
<!– We are adding new items to popup menus -->
<extension point="org.eclipse.ui.popupMenus">
      <!– Add a new item to popup menus for files -->
      <objectContribution adaptable="false id="demos.eclipse.plugins.types.fileContribution"
                              objectClass="org.eclipse.core.resources.IFile">
            <action class="demos.eclipse.plugins.types.FileAction"
                  id="demos.eclipse.plugins.types.files" label="Trigger File Action"/>
      </objectContribution>
      <!– Add a new item to popup menus for Folders -->
      <objectContribution adaptable="false" id="demos.eclipse.plugins.types.folderContribution"
                              objectClass="org.eclipse.core.resources.IFolder">
            <action class="demos.eclipse.plugins.types.FolderAction"
                  id="demos.eclipse.plugins.types.folders" label="Trigger Folder Action"/>
      </objectContribution>
      <!– Add a new item to popup menus for Java Packages -->
      <objectContribution adaptable="false" id="demos.eclipse.plugins.types.packageContribution"
                  objectClass="org.eclipse.jdt.core.IPackageDeclaration">
            <action class="demos.eclipse.plugins.types.PackageAction"
                  id="demos.eclipse.plugins.types.packages" label="Trigger Package Action"/>
      </objectContribution>
</extension>
```

Commonly Used Extension Points

Extension Point ID	Purpose
org.eclipse.ui.popupMenus	Adding new items to popup menus
org.eclipse.ui.views	Adds new views to a workbench page
org.eclipse.ui.actionSets	Adds menus, menu items or toolbar buttons
org.eclipse.ui.editors	Adds a new editor (typically for files)
org.eclipse.ui.preferencePages	Contribute a new page for a type contained within the preferences GUI found under 'Window->Preferences'
org.eclipse.ui.propertyPages	Contributes a page to be displayed when you right-click on a type and select 'properties'
org.eclipse.ui.newWizards	Adds a wizard triggered through the 'File->New' menu
org.eclipse.ui.importWizards	Adds a wizard triggered through the 'File->Import' menu

Base Types Used for Extensions

Extension Point Type	Base Class and/or Interface
org.eclipse.ui.popupMenus	org.eclipse.ui.IObjectActionDelegate
org.eclipse.ui.views	org.eclipse.ui.part.ViewPart
org.eclipse.ui.actionSets	org.eclipse.ui.IWorkbenchWindowActionDelegate
org.eclipse.ui.editors	org.eclipse.ui.IEditorPart org.eclipse.ui.part.EditorPart
org.eclipse.ui.preferencePages	org.eclipse.ui.IWorkbenchPreferencePage
org.eclipse.ui.propertyPages	org.eclipse.ui.IWorkbenchPropertyPage
org.eclipse.ui.newWizards	org.eclipse.ui.INewWizard org.eclipse.jface.wizard.Wizard
org.eclipse.ui.importWizards	org.eclipse.ui.IImportWizard org.eclipse.ui.dialogs.WizardImportPage

The Plug-In Class

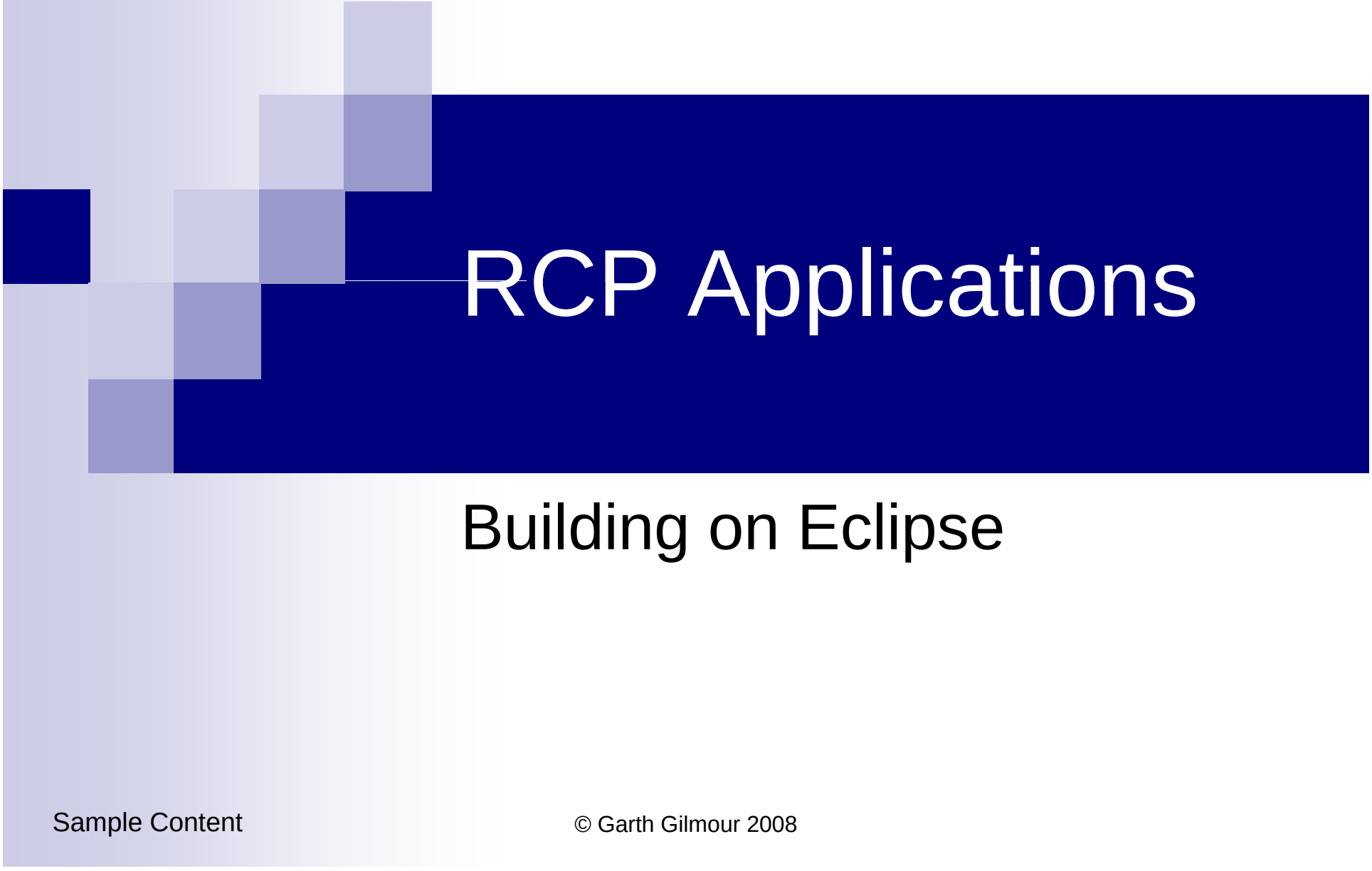
- A plug-in optionally has a plug-in class
  - This represents the plug in as a whole
  - When the plug-in is used for the first time this class will be the first loaded from the java archive
  - Developers typically use a plug-in class to manage resources
- The plug-in class inherits from ‘AbstractUIPlugin’
  - This defines ‘start’ and ‘stop’ lifecycle methods
  - If the ‘start’ method throws an exception the plug-in will be considered has having failed and will not be reloaded
- Eclipse guarantees the plug-in class is a singleton
  - It only ever creates a single instance of the class
  - This can be obtained by calling the ‘getDefault’ method

Working With Java Code in Eclipse

- Eclipse has its own library for representing code
  - Using the classes found in ‘org.eclipse.jdt.core’
  - The ‘IType’ interface represents a class or interface
  - The ‘ICompilationUnit’ interface represents a ‘.java’ file
- A plug-in can access and alter these structures
  - There are several different ways of doing this
- Changes should be made on a ‘working copy’
  - Calling ‘getWorkingCopy’ on a compilation unit makes a copy
  - This can be merged with the original by calling ‘reconcile’
  - You can overwrite the original via ‘commitWorkingCopy’

Working With Java Code in Eclipse

- Simple changes can be made directly
  - By calling one of the modifier methods of ‘IType’
  - E.g. calling ‘createMethod’ adds a new method to the corresponding class or interface
- Complex changes require accessing the source code
  - By calling ‘compilationUnit.getBuffer().getCharacters()’
- There are two utilities to help alter source code
  - An ‘IScanner’ tokenizes the source for you
  - An ‘ASTParser’ converts the source into an abstract syntax tree
    - Which can then be navigated and manipulated
    - The visitor design pattern simplifies the process



Building on Eclipse

Sample Content

Eclipse Rich Client Platform (RCP)

- RCP was an unintended result of Eclipses popularity
  - Swing based GUI development had gained a bad reputation
  - Development teams wanted to use the Eclipse runtime and workbench as the framework for their project
- In response to this the code for the Java IDE was separated from the core Eclipse framework
  - From Eclipse 3 onwards you can use the core framework for building non-IDE based GUI's
  - You can even use the runtime in isolation for server-side applications

Eclipse Rich Client Platform (RCP)

- A RCP project is similar to a normal plug-in
  - Except that instead of contributing single items you start with an empty workbench and contribute an entire perspective
- Eclipse now provides a set of wizards for generating skeleton applications which you can then extend
  - This is helpful as there is a lot of ‘plumbing’ involved in RCP development…
- Editors, views etc… are still specified in ‘plugin.xml’ as extensions but you have to link them together
  - Eclipse provides a core infrastructure and component set

Configuring a Rich Client Project

- To create a product based on Eclipse declare an extension with id ‘org.eclipse.core.runtime.applications’
  - This is how the Eclipse Java IDE is itself defined
  - A single class must be defined which starts the product

```
<extension id=“application” point="org.eclipse.core.runtime.applications">
      <application>
            <run class="demos.eclipse.rcp.hello.Application“/>
     </application>
</extension>
<extension point="org.eclipse.ui.perspectives">
      <perspective
                  name="Hello Perspective"
                  class="demos.eclipse.rcp.hello.Perspective"
                  id="demos.eclipse.rcp.hello.perspective">
     </perspective>
</extension>
```

Initializing an RCP Based Project

- The main class implements ‘IPlatformRunnable’
  - This defines a ‘run’ method which acts as the entry point for the application (the RCP equivalent of ‘main’)
- Inside the ‘run’ method you must:
  - Obtain the current SWT ‘Display’ object
    - Via ‘PlatformUI.createDisplay’
    - This manages the interaction between SWT and the OS
  - Initialise and run your workspace
    - Via ‘PlatformUI.createAndRunWorkbench’
      - This takes the current display object and a ‘WorkbenchAdvisor’
      - It handles the SWT event handling loop on your behalf

Initializing an RCP Based Project

```
public class Application implements IPlatformRunnable {

	public Object run(Object args) throws Exception {
		Display display = PlatformUI.createDisplay();
		try {
			WorkbenchAdvisor advisor = new ApplicationWorkbenchAdvisor();
			int returnCode = PlatformUI.createAndRunWorkbench(display,advisor);
			if (returnCode == PlatformUI.RETURN_RESTART) {
				return IPlatformRunnable.EXIT_RESTART;
			}
			return IPlatformRunnable.EXIT_OK;
		} finally {
			display.dispose();
		}
	}
}
```

Initializing an RCP Based Project

- The workbench advisor provides:
  - The id of the perspective for your application
  - A ‘WorkbenchWindowAdvisor’ object which:
    - Contains callback methods for the lifecycle of the window
    - Supplies another manager object that controls action bars

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
	private static final String PERSPECTIVE_ID = "demos.eclipse.rcp.hello.perspective";
	public String getInitialWindowPerspectiveId() {
		return PERSPECTIVE_ID;
	}
	public WorkbenchWindowAdvisor
			createWorkbenchWindowAdvisor(IWorkbenchWindowConfigurer configurer) {
		return new ApplicationWorkbenchWindowAdvisor(configurer);
	}
}
```

Deploying and Updating RCP Apps

- Deploying plug-ins and RCP apps is simple
  - They can be zipped or use a native installer like ‘InstallShield’
- Updating applications is more complex
  - Eclipse includes an updating component that can be used to install or update features
  - Components are downloaded from an ‘Update Site’
    - This can be mounted across the network or on a local drive
- One way to avoid these problems is ‘WebStart’
  - This provides a standard VM on the client machine and the ability to download applets and applications with smart caching
    - Classes are cached but new versions are automatically downloaded
    - The application is described by a ‘.jnlp’ file on the server