Patterns for Single-Sourcing RCP and RAP applications

Benjamin Muskalla (EclipseSource)

Version: **1.1.4**

Abstract

Developing fully single-sourced bundles for Eclipse RCP and RAP requires following certain practices. This paper describes different patterns and refactorings to achieve single sourcing of applications for both runtime platforms. These patterns have been gleaned from our experiences single sourcing various applications in the Eclipse.org runtime space and tooling space.

Contents

1 Introduction

The purpose of the Eclipse RT top-level project at Eclipse is to bring together various runtime related efforts and technologies and to foster, promote and house runtime efforts at Eclipse[1]. It is part of a larger Equinox Community drive to implement Equinox-based technology across a broad range of computing environments and problem domains. Underlying all these efforts is the common goal of providing a uniform component model across a wide variety of computing environments. The Equinox framework and OSGi [All03] form the basis of this infrastructure.

Write once, run everywhere is the main objective of the Java™programming language [AHL+05]. This holds true most of the time for running the same program on different operating systems. But as technology evolves the Eclipse Foundations strives to bring this slogan to a new level. Two technology projects, housed under the Eclipse.org umbrella, namely RAP[2] and eRCP[3], provide an alternative runtime environment for RCP applications. This means that the application - normally running as a desktop application on a personal computer - can now be deployed to other runtime environments. For example let us assume we have a ready to launch application based on RCP. By switching the runtime from RCP to RAP, it becomes possible to launch the application on an application server where clients ccan use the application with a Web 2.0 centric interface on a browser. There is no need for the user to install any further add-ons or plugins. In order to achieve full compatibility between the platforms, many concepts implemented in SWT [NW04] need to be adapted to other runtimes. These are hidden behind the public API which remains synchronous across all runtime projects.

Developing fully single-sourced bundles for Eclipse RCP and RAP requires following certain practices. It may also be possible to change the structure of the application code to fit the current runtime environment. Depending on the specific case, this may be achievable through normal refactorings [FBB+99]. The refactorings are not onerous and generally lead to better quality bundles irrespective of the single-sourcing requirement.

[1] http://www.eclipse.org
[2] http://www.eclipse.org/rap
[3] http://www.eclipse.org/ercp

2 Workspace organization

2.1 Target platforms

Eclipse PDE uses the concept of a "Target Platform" to manage the set of third-party bundles available for development. The target platform is the directory containing the pre-built binary bundles that the workspace bundles depend on. It is used to compile the workspace and is also used when the user launches an application by clicking Run or Debug from the workbench. RAP itself delivers its own target platform so developers can easily switch between the RCP plugins and the RAP counterparts.

By default Eclipse uses its own installation directory as the Target Platform, which is suitable for developing SDK plug-ins but is less suitable for developing RCP or RAP bundles. For RCP, this default Target Platform is unsuitable because it contains many superfluous bundles relating to the IDE functionality, which may inadvertently be pulled in as dependencies of the application. For RAP the default is unsuitable for the same reason, and in addition it doesn't contain all the RAP-specific API bundles.

Managing two separate target platforms is currently one of the key challenges when single-sourcing applications for RAP and RCP. Unfortunately PDE only allows one Target Platform at a time. There are two disadvantages to this limit. First, switching the target is very time-consuming as the whole workspace needs to be recompiled. Secondly, we need to close all related projects which do not match the target platform in order to have a clean workspace. A reasonable solution which has been become apparent is to use different workspaces for the different runtime environments. However, it is important to frequently check the code for compliance with the alternative runtime environments. An automated build should be used to perform a full build against both Target Platforms at regular intervals.

2.2 Using separate workspaces

In order to manage the two target platforms, we create two separate workspaces. Generally within a given workspace we have two types of projects. First, projects that are common for both platforms which can be reused without any modification. Secondly, we have projects that are platform-specific and thus contain only code that runs on one of the platforms (eg. an entrypoint implementation). By utilizing separate workspaces for each platform we can import all common projects into both workspaces and have the projects unique for one platform in the corresponding workspace. A regular layout of workspaces is the following:

1. RCP Workspace
   - Common projects
   - RCP-specific projects
   - *RCP target platform*
2. RAP Workspace
   - Common projects
   - RAP-specific projects
   - *RAP target platform*

As the common projects should be shared across both workspaces you need to ensure that they share the same files on the file system. The best way to achieve this is to create all projects in the same folder and import them into your workspaces. When you import the projects, be sure not to check the "Copy projects into workspace" option.

File system

- Common projects

- RCP-specific projects
- RAP-specific projects

2.3 Develop principally on the RAP Target Platform

For effective single-sourcing we should only use API elements that are available in **both** RCP and RAP. Unfortunately, there is no collection of bundles available that contains a pure subset of both, as shown in Figure 1. As we need to choose one or the other it is recommended to develop on RAP. The reason is that the set of APIs offered by RAP is smaller than the set of APIs offered by RCP. Also it tends to be more obvious when one is using a RAP-specific API.

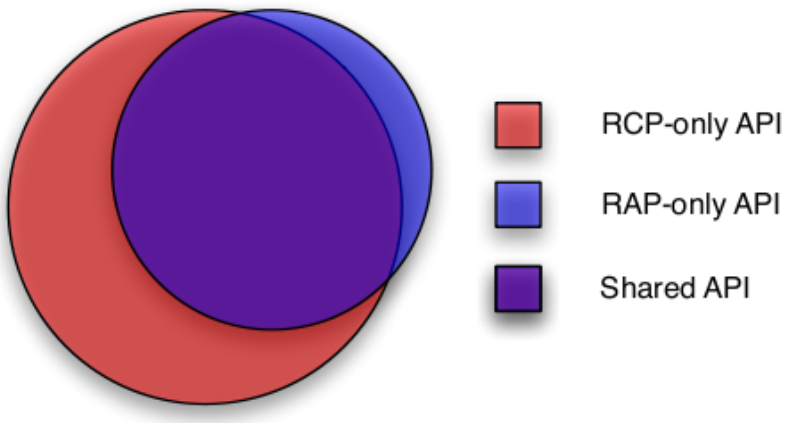

Figure 1: The RCP and RAP APIs

3 Dependencies

3.1 The problem - What strategy is best for managing dependencies

OSGi (and therefore Eclipse RCP and RAP) offers two separate mechanisms for declaring dependencies between bundles: Import-Package, which provides dependencies on the Java package-level, and Require-Bundle, which provides for dependencies based on whole bundles.

3.1.1 Bundle dependencies

Require-Bundle (*Require-Bundle*) was introduced in OSGi Release 4, principally to support legacy Eclipse plug-ins, which in Eclipse 3.0 became OSGi bundles. The pre-3.x Eclipse module system did not have any concept similar to *Import-Package*, and dependencies were specified based on the ID of another plug-in. Therefore in OSGi R4, *Require-Bundle* works by importing all of the packages exported from a particular named bundle. *Require-Bundle* still tends to be favored by many Eclipse plug-in developers both for historical reasons and because PDE makes it significantly easier to use in development than *Import-Package*.

3.1.2 Package dependencies

Import-Package is the original mechanism used in OSGi, and until Release 4 it was the only one. It is very simple: a bundle lists all of the Java packages that it imports from other bundles, including packages supplied by the JRE such as javax.swing, org.w3c.dom etc. These packages are then resolved at runtime by the OSGi framework (i.e., Equinox) against a corresponding exported package from another bundle.

3.1.3 Split packages

The eclipse workbench uses the OSGi concept of "split packages". This enables OSGi to have virtual packages which are physically split across several bundles. See the OSGi 4.1 specification (§3.13.3) for more informations about split packages.

In order to import only a specified part of the package, you need to extend the Import-Package declaration with the split attribute as shown in the following MANIFEST.MF fragment:

```
Manifest-Version: 1.0

Import-Package: org.eclipse.ui; ui.workbench="split",
                org.eclipse.ui.part; ui.workbench="split"
```

Listing 1: Importing a split package

The ui.workbench="split" directive tells Equinox to use only the "ui.workbench" part of this split package. Otherwise the dependencies would not be resolved.

3.2 Favor Require-Bundle

Even though *Import-Package* is recommended by OSGi developers it has the drawback of split packages. In addtion, the tooling support in PDE is much better for *Require-Bundle*.

In order to avoid hving to manage the dependencies in every bundle, you should create a new bundle (eg. com.foo.bar.ui.compatibility) which has (optional) dependencies on both runtime bundles.

```
Manifest-Version: 1.0

Require-Bundle: org.eclipse.rap.ui;resolution:=optional;
                  visibility:=reexport,
                org.eclipse.rap;resolution:=optional;
                  visibility:=reexport
```

Listing 2: Optional bundle requirements

Now all the application's UI bundles can use the compatibility bundle as their main dependency for getting "wired" to the correct bundle at runtime.

4 Missing API

4.1 The problem

Inevitably some parts of an application cannot be single-sourced. For example we may decide to offer convenience features in the RCP edition that are not available under RAP, or develop presentation or "theming" code for the RAP edition. In each case this leads to code that will compile under RCP but not RAP and *vice versa*.

4.2 Building fragments

It is important to sort the runtime-specific code into separate "fragments". Fragments are incomplete bundles (hence the name) which attach themselves to other bundles and extend them in some way. If RCP-specific code exists in a bundle then that whole bundle will not be resolved in RAP, including generic code that might otherwise have worked in both environments. By making separate fragments for the RCP-only or RAP-only code, we can maximise single-sourcing. Furthermore, in some cases it is useful to separate UI code from "core" or non-UI code, so that the non-UI code can be reused elsewhere. In general, a bundle should contain a logical grouping of business functionality. With our approach, there will be up to one bundle and one fragment for each such grouping. The naming should follow a standard scheme as shown:

Table 1: Bundle Naming Scheme

Bundle/Fragment Name	Description
com.foo.bar.ui	UI (both RCP and RAP) functionality
com.foo.bar.ui.rcp	RCP-only functionality *(fragment)*
com.foo.bar.ui.rap	RAP-only functionality *(fragment)*

Which parts of the application will live in the host bundle and what is platform specific will be discussed in the rest of this paper.

4.3 Extract a compatibility plug-in

Depending on the requirements, it may be appropriate to extract a bundle which contains platform-specific differences which are not bound to the current application. We use the term compatibility plugin to refer to this bundle. It has the same fragment structure as discussed above. The compatibility bundle can be reused in future development. See also Figure 2 for an overview of this approach.

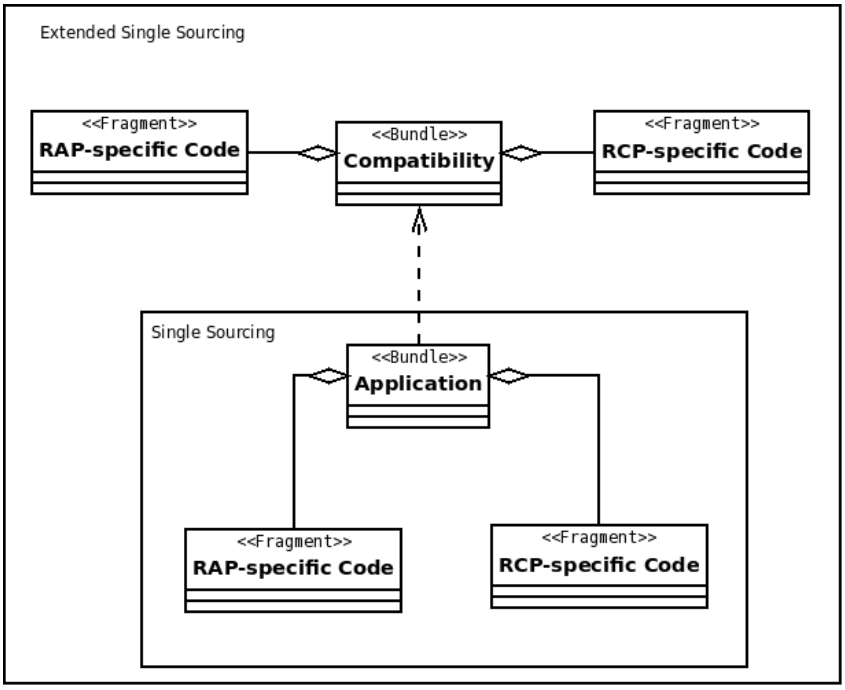

Figure 2: Extended single sourcing

5 Missing extension points

5.1 The problem

Some of the extension points used in regular RCP applications are not available in RAP. For example to the *org.eclipse.ui.bindings* extension point is not available. On the other hand there are contributions which are specific to RAP such as *themes*, *brandings* or the *entrypoint* extensions.

5.2 Move extensions to fragments

Since we already have defined fragments we can move the platform-specific extensions to the *fragment.xml* of the corresponding fragment projects. This way the extensions are only contributed to the runtime when the fragment is available and thus only when we have the right runtime.

For an example of a *fragment.xml* for the RAP-specific fragment, see the following snippet.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<fragment>
   <extension
         point="org.eclipse.rap.ui.entrypoint">
      <entrypoint
            class="maildemo.EntryPoint"
            id="maildemo.rap.entrypoint"
            parameter="maildemo">
      </entrypoint>
   </extension>

</fragment>
```

Listing 3: Sample fragment.xml

6 Application startup and Activator scope

6.1 The problem

A widely used approach to initialize an RCP application is the use of Activators that are called while starting an OSGi bundle. See the OSGi 4.1 specification (§4.3.6) for more information about Activators.

While initializing the needed artifacts during bundle start is a good approach in RCP - it can lead to serious problems in a multi-user environment like RAP.

One of the key RAP behaviours during runtime is that all OSGi bundles are started only **once**. Not every user of the application has its own set of bundles - instead all bundle instances are shared across all users (and sessions). This means that the Activator of a bundle has two semantic differences:

1. The Activator is called *once* per application, not per user
2. It is called in *application scope*, not in session scope

6.2 Separate session-specific code from Activator

As there can be no active session when a bundle starts we need to separate all session and user related code from the Activator so it can be executed when the session starts.

```
public class Activator extends AbstractUIPlugin {

  public void start(BundleContext context) throws Exception {
    super.start(context);
    // session specific code
  }

}
```

Listing 4: Examplary Activator

One solution is to move the related code into the *createUI* method of the entrypoint as this will be executed in the current session context.

```
public class EntryPoint implements IEntryPoint {

  public int createUI() {
    // session specific code|
    Display display = PlatformUI.createDisplay();
    return PlatformUI.createAndRunWorkbench(display, new ApplicationWorkbenchAdvisor());

  }

}
```

Listing 5: Examplary entrypoint

Another possibility is to use the *org.eclipse.ui.startup* extension point to contribute new runnables into the startup process of the workbench. These *IStartup* implementations are registered trough the extension point mechanism as shown in the following snippet:

```
<extension point="org.eclipse.ui.startup">
     <startup class="org.eclipse.example.StartupClass"/>
</extension>
```

Listing 6: plugin.xml

This is optimal for situations such as when the workbench has a *session-scope* and the *IStartup* implementations are to be called when the session starts.

7 API Differences

7.1 The problem

Inevitably there are some differences between the two runtimes - a common problem when single sourcing an application. RAP provides a strict subset of all available SWT and RCP APIs. If the API is available it has the same name, belongs to the same package and behaves the same way as the corresponding RCP API. But this means that API which is not available in RAP needs to be wrapped somehow in order to be able to use it in the common codebase. Some examples for missing API are:

- GC
- FileDialog
- MouseMove Events

However, RAP itself provides a minimal set of APIs which are not part of RCP. These were introduced in order to meet several web specific requirements. The following describes how to use these APIs without breaking the RCP implementation.

7.2 Encapsulate the problem

The first step to get around these obstacles is to provide a common interface to access the required artifact. What the artifact really is depends on the API usage of the application. It could be a simple object, an action or even whole dialogs.

To have a common accessor to those artifacts we need to create an abstract type in the host bundle that will provide us later with the concrete implementation.

As a concrete example we take the *ActionFactory* of the workbench. It defines several preconfigured actions to be integrated into the application. One of these actions - the *ABOUT* action - is not available in RAP.

Using the *ABOUT* in RCP applications is not unusual. The following is a snippet taken from an *ActionBarAdvisor* of an RCP application.

```
aboutAction = ActionFactory.ABOUT.create(window);
register(aboutAction);
```

Listing 7: About Action

As *ActionFactory.ABOUT* is not available we need to replace it with a facade. This facades delegates the actual request during the runtime to the corresponding fragment project. In the case of RCP we can use the real *ActionFactory.ABOUT* implementation. Providing an alternative action for the web is done by returning another action implementation by the RAP fragment. We replace the real API with a call to our new facade which follows in the next paragraph.

```
aboutAction = ActionFactoryFacade.createAboutAction(window);
register(aboutAction);
```

Listing 8: Replace the call with the facade

Then we create the facade we have introduced.

```
public abstract class ActionFactoryFacade {
  private final static ActionFactoryFacade IMPL;
  static {
    IMPL = ( ActionFactoryFacade ) ImplementationLoader.newInstance(
        ActionFactoryFacade.class );
  }

  public static IWorkbenchAction createAboutAction( final IWorkbenchWindow window ) {
    return IMPL.createAboutActionInternal( window );
  }

  abstract IWorkbenchAction createAboutActionInternal( IWorkbenchWindow window );
```

```
}
```

Listing 9: Abstract type in host bundle

The purpose of the facade is to load the correct implementation of the ActionFactoryFacade and call their internal methods to create the requested objects. The *ImplementationLoader* is therefore needed to load the implementation of the abstract type from the fragment. It should be implemented as the following snippet shows:

```
public class ImplementationLoader {

  public static Object newInstance(final Class type) {
    String name = type.getName();
    Object result = null;
    try {
      result = type.getClassLoader().loadClass(name + "Impl").newInstance();
    } catch (Throwable throwable) {
      String txt = "Could not load implementation for {0}";
      String msg = MessageFormat.format(txt, new Object[] { name });
      throw new RuntimeException(msg, throwable);
    }
    return result;
  }
}
```

Listing 10: ImplementationLoader

We now have a facade that encapsulates the required functionality and the *ImplementationLoader*. Our final step is to create two different implementations of the behavior for the different runtimes.

As mentioned above the implementation for the *ABOUT* action can simply be delegated to the original RCP equivalent.

```
public class ActionFactoryFacadeImpl extends ActionFactoryFacade {

  IWorkbenchAction createAboutActionInternal( IWorkbenchWindow window ) {
    return ActionFactory.ABOUT.create( window );
  }

}
```

Listing 11: Concrete type for RCP fragment

For RAP we need to provide our own implementation of the *ABOUT* action here.

```
import org.eclipse.jface.action.Action;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.actions.ActionFactory.IWorkbenchAction;

public class ActionFactoryFacadeImpl extends ActionFactoryFacade {

  private class AboutAction extends Action implements IWorkbenchAction {
    private IWorkbenchWindow window;
    public AboutAction(IWorkbenchWindow window) {
      this.window = window;
      setId( "about" );
      setText("About RAP MailDemo");
      setToolTipText("About RAP MailDemo");
    }
    public void dispose() {
      window = null;
    }
    public void run() {
      String title = "About Message";
      String msg = "This is the about Message of the RAP Mail Demo";
      MessageDialog.openInformation(window.getShell(), title, msg );
    }
  }

  IWorkbenchAction createAboutActionInternal(IWorkbenchWindow window) {
    return new AboutAction(window);
  }
}
```

Listing 12: Concrete type for RAP fragment

8 Field validation

8.1 The problem

A common pattern for form validation in RCP and SWT applications is to use the *ModifyListener* interface to implement keystroke-level validation of field data. This has the advantage of producing more immediate feedback for the user regarding the data they are entering. The following code snippet shows an example of this technique:

```
Text txtDate = new Text(parent, SWT.BORDER);
txtDate.addModifyListener(new ModifyListener() {
  public void modifyText(ModifyEvent e) {
    try {
      date = formatter.parse(txtDate.getText());
      setErrorMessage(null);
    } catch (ParseException e) {
      setErrorMessage("Invalid date");
    }
  }
});
```

Listing 13: Usage of ModifyListener

Unfortunately the *Modify-* and *VerifyListener* of RWT do not behave exactly as their correspoinding parts in SWT. This can be traced back to the distributed nature of RAP. Over a high-latency network connection the SWT behavior can introduce significant performance degradation to the UI. RWT generally reduces the number of requests sent to the server and decreases server load by merging multiple events into one HTTP request. This is especially interesting when utilizing the *doit* flag of the incoming event as this may match to several keystrokes instead of a single keystroke. This can lead to semantic differences as the following example illustrates. Imagine that we have a VerifyLister that restricts the length of a text field.

```
final Text t = new Text( shell, SWT.BORDER SWT.MULTI );
t.addVerifyListener( new VerifyListener(){

  public void verifyText( VerifyEvent event ) {
    if( t.getText().length() > 5 ) {
      event.doit = false;
    }
  }

});
```

Listing 14: Example of semantic differences

With SWT this snippet works as expected. Running the same snippet with RWT you need to be careful as the events are merged into bigger chunks. This can lead to a problem when you type more than 5 characters into the textfield. Depending on the timing it could happen that the *doit* flag will be set to *false* for the whole chunk - not only for the events after the fifth character.

The Eclipse Databinding framework is, in most situations, based on *VerifyListener*s and thus is subject to the same restrictions. In the case of a high-latency network connection you should be careful when integrating the Databinding framework.

8.2 Reduce the frequency of validation checks

Depending on the latency of the network it may be a better approach to reduce the frequency of validations. This can be achieved by using listeners which instead of being triggered by keystrokes are triggered, for example, by focus events.

```
  final Text t = new Text( shell, SWT.BORDER SWT.MULTI );
  t.setText( "here is some text" );
  t.addFocusListener( new FocusAdapter(){

    public void focusLost( FocusEvent event ) {
```

```
      if( t.getText().length() == 0 ) {
        t.setBackground( Graphics.getColor( 255, 0, 0 ) );
      } else {
        t.setBackground( null );
      }
    }

  });
```

Listing 15: FocusListener for field validation

With this approach the events only get fired on focus events no matter what happens inside the text field.

9 SWT Resources

9.1 The problem

As RAP lives in a server environment the logic is only run on the server side while the client is responsible for showing the UI. Thus the concepts of SWT for acquiring system resources like colors or fonts cannot be carried over in exactly the same fashion. RAP also lowers memory consumption on the server-side by reusing existing resources. Therefor RAP banned constructors from all SWT resources while providing factory methods for the following resource classes.

- org.eclipse.swt.graphics.**Color**
- org.eclipse.swt.graphics.**Font**
- org.eclipse.swt.graphics.**Image**
- org.eclipse.swt.graphics.**Cursor**

When writing SWT code you are responsible for disposing all resources created in your code. Under normal circumstances this is done by calling *dispose* on the resource object. As this would invalidate the resources across all sessions, RAP does not provide the dispose mechanism at all.

9.2 Use higher-level API

A pattern that frequently occurs is the use of use an abstraction layer above these APIs. In the case of resources you can use the **Registry* classes that JFace already provides. Instead of managing the resources yourself you let the registry do the work. Neither the instantiation nor the recycling of resources needs to be done manually. The big advantage is that this approach can be used in RCP *and* RAP so there is no need to have a different implementations.

Starting with a simple SWT snippet which acquires a new color object, we will implement the same behavior with the use of a *ColorRegistry*.

```
import org.eclipse.swt.graphics.Color;

Color redColor = new Color( display, 255, 0, 0 );
```

Listing 16: Creating a new color (SWT)

Instead of initiating the color object directly,we manage the lifetime of these objects with registries that JFace provides.

```
import org.eclipse.jface.resource.ColorRegistry;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.RGB;

ColorRegistry registry = new ColorRegistry( Display.getCurrent() );
registry.put( ``redColor'', new RGB( 255, 0, 0 ) );
Color redColor = registry.get( ``redColor'' );
```

Listing 17: Creating a new color (JFace)

There may be situations when it is not possible to use the JFace registries. The above mentioned resources can also be obtained directly through the factory methods of RAP. These are available on the *Graphics* class introduced by RAP.

```
import org.eclipse.swt.graphics.Color;

import org.eclipse.rwt.graphics.Graphics;

Color redColor = Graphics.getColor( 255, 0, 0 );
```

Listing 18: Creating a new color (RWT) - not recommended

As the *Graphics* class is only available in RAP, the corresponding source artifact needs to be added to the RAP specific fragments. For the RCP implementation constructor would be used.

An important item to note is that the resources in RCP need to be disposed manually. This results in an empty implementation for the RAP fragment.

10 Singletons and Scopes

10.1 The problem

As RAP is a multi-user and distributed runtime we need to make sure that everything which originally runs in a single-user mode will have the same semantics in the multi-user environment.

One of the most common problems is the usage of the Singleton design pattern. [GHJV95] This pattern allows only one unique object instance of a class. As RAP needs to serve several users at the same time this may not be the needed approach. Most singletons should be unique per user. In contrast to RCP, with RAP we have not only one but an unlimited number of users.

10.2 Use session singletons

To make singletons aware of the user scope we need to restrict the singletons to a specific user session instead of one application-wide object.

A common implementation of an application-scoped singleton in RCP is as follows:

```
public class MySingleton {
  private static MySingleton instance;

  public static MySingleton getInstance() {
    if( instance == null ) {
      instance = new MySingleton();
    }
    return instance;
  }

  private MySingleton() {
    // prevent instance creation
  }

}
```

Listing 19: Singleton (Application scope)

To reduce the scope of the object instance to a specific session we use the concept of '*session singletons*'. RAP already provides a helper class to create objects like this within the session scope. The class is called *SessionSingletonBase* and is used as follows:

```
public class MySingleton {

  public static MySingleton getInstance() {
    return ( MySingleton.class )SessionSingletonBase.getInstance( MySingleton.class );
  }

  private MySingleton() {
    // prevent instance creation
  }

}
```

Listing 20: Singleton (Session scope)

Doing this behind a facade we can reuse the same concept as we used in chapter 7.

10.3 Move the instance creation to the fragments

The instance creation is delegated to one of the fragment implementations by a common accessor. As the fragments need to have the same interface for creating the instance we use an interface called *ISingletonProvider*. Both the singleton and the provider interface should be created in the host bundle.

```
public class MySingleton {

  private static final ISingletonProvider PROVIDER;
  static {
    PROVIDER = ( ISingletonProvider ) ImplementationLoader.newInstance( MySingleton.class );
  }
```

```

  public static MySingleton getInstance() {
    return ( MySingleton ) PROVIDER.getInstanceInternal();
  }

  private MySingleton() {
    // prevent instance creation
  }

}
```

Listing 21: Singleton (delegating stub)

```
public interface ISingletonProvider {
  Object getInstanceInternal();
}
```

Listing 22: Singleton Provider Interface

As the singleton provider differs between the platform-specific fragments (singleton for RCP, session-singleton for RAP) we need to provide two different implementations of the interface.

For RCP we implement the *ISingletonProvider* interface as a regular singleton. This implementation belongs to the RCP-fragment.

```
public class MySingletonImpl {

  private static MySingleton instance;

  public synchronized Object getInstanceInternal() {
    if( instance == null ) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

Listing 23: Singleton Provider (RCP)

When running the application with multiple users, we want to create a session-scoped singleton. This implementation belongs to the RAP-fragment.

```
public class MySingletonImpl {

  public Object getInstanceInternal() {
    return SessionSingletonBase.getInstance( MySingleton.class );
  }

}
```

Listing 24: Singleton Provider (RAP)

With this architecture we are able to reuse the same code in RAP and RCP. The singleton implementation lives in the host bundle and only the way it is initiated is platform-specific.

11 Jobs and background threads

11.1 The Problem

Utilizing jobs[4] in RCP and RAP applications is a common pattern to process asynchronous work in the background. However, to access something session-specific like a session singleton you need to be careful. As background jobs run in the application scope and not in the session scope we need to provide the correct context for accessing session-specific objects.

11.2 Use a facade

The approach is the same as in chapter 7. We work against an abstract facade in our host bundle and provide different implementations in each fragment.

The facade is just a factory to return a new *Job* instance.

```
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.swt.widgets.Display;

public abstract class JobFactory {
  private final static JobFactory IMPL;
  static {
    IMPL = (JobFactory) ImplementationLoader.newInstance(JobFactory.class);
  }

  public static Job createJob( final Display display, final String name, final JobRunnable
      runnable) {
    return IMPL.createJobInternal(display, name, runnable);
  }

  abstract Job createJobInternal( Display display, String name, JobRunnable runnable);
}
```

Listing 25: Abstract Job Factory

To encapsulate the code that will be run in a job we introduced a *JobRunnable*. This has the same semantics as a regular *java.lang.Runnable* but has the ability to transfer the status of the job (see *IStatus*).

```
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;

public interface JobRunnable {
  IStatus run(IProgressMonitor monitor);
}
```

Listing 26: JobRunnable

As RCP assumes only one user (session) we can execute the runnable in a regular job. The implementation for the RCP fragment is shown below.

```
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.swt.widgets.Display;

public class JobFactoryImpl extends JobFactory {

  Job createJobInternal(final Display display, final String name, final JobRunnable
      runnable) {
    return new Job(name) {
      protected IStatus run(IProgressMonitor monitor) {
        return runnable.run(monitor);
      }
    };
  }

}
```

Listing 27: Job Factory Implementation (RCP)

[4] http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html

In the case of RAP we need to process the runnable inside an environment with a context. To do this RWT provides the method *runNonUIThreadWithFakeContext*.

```java
package facades;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.rwt.lifecycle.UICallBack;
import org.eclipse.swt.widgets.Display;

public class JobFactoryImpl extends JobFactory {

  Job createJobInternal(final Display display, final String name, final JobRunnable
      runnable) {
    Job job = new Job(name) {
      protected IStatus run(final IProgressMonitor monitor) {
        final IStatus[] result = new IStatus[1];
        UICallBack.runNonUIThreadWithFakeContext(display,new Runnable() {
          public void run() {
            result[0] = runnable.run(monitor);
          }
        });
        return result[0];
      }
    };
    return job;
  }
}
```

Listing 28: Job Factory Implementation (RAP)

12 Internationalization and localization

12.1 The problem

In RAP we have to deal with different languages for different user sessions. In fact, the language can also change between requests within the same session. Therefore, we cannot store language related information statically in *Message Bundle* classes as it is stored in RCP. Instead, we have to use a different instance of the *Message* class for every language. The *NLS* class is part of the regular RCP internationalization mechanism [5] and follows the usage conventions of the OSGi 4.1 specification (see §4.3.6).

An example of such a message class can be found in the following.

```
import org.eclipse.osgi.util.NLS;

public class Messages extends NLS {
  private static final String BUNDLE_NAME = "maildemo.messages"; //$NON-NLS-1$

  public static String OpenViewAction_0;
  public static String OpenViewAction_1;
  public static String OpenViewAction_2;
  public static String OpenViewAction_3;

  static {
    // initialize resource bundle
    NLS.initializeMessages(BUNDLE_NAME, Messages.class);
  }

  private Messages() {
  }
}
```

Listing 29: Sample RCP Message class

12.2 Make translatable strings non-static

The first thing we need to do is remove the static modifier from all translatable messages. Otherwise the static messages would be shared across all sessions and we could not provide an internationlized application for each user. In our initial example we need to remove the *static* keyword from all *OpenViewAction* fields.

12.3 Remove static initializer and supertype

As with the *static* keyword for all message fields, we remove the static initializer from all translatable messages to avoid initializing the contents of the message fields when the class is loaded.

In order to use our own NLS mechanism we also need to ensure that the message class does **not** extend the *NLS* class of OSGi.

We also have to provide an accessor to get an instance of our message class which is localized during runtime. This will just delegate the real work to a helper (eg. *NLSHelper*) which in turn gives us an instance of the class with all fields initialized with the corresponding strings.

```
  public static Messages get() {
    return NLSHelper.getMessages(Messages.class);
  }
```

Listing 30: Accessor for Message classes

[5] http://help.eclipse.org/ganymede/topic/org.eclipse.jdt.doc.user/concepts/concept-string-externalization.htm

12.4 Create an NLSHelper and implementations

The concept is the same as in chapter 7. We work against an abstract facade in our host bundle and provide different implementations in each fragment. On our host bundle we have the abstract class which loads to platform-specific implementation from one of the fragments.

```
public abstract class NLSHelper {

  protected static final String BUNDLE_NAME = "sample.messages"; //$NON-NLS-1$

  private final static NLSHelper IMPL;
  static {
    IMPL = (NLSHelper) ImplementationLoader.newInstance(NLSHelper.class);
  }

  public static Messages getMessages(Class clazz) {
    return (Messages) IMPL.internalGetMessages(clazz);
  }

  protected abstract Object internalGetMessages(Class clazz);
```

Listing 31: NLS Helper Facade

RAP itself provides some helper classes in order to fully support multi-language applications. One of these classes is *RWT.NLS* which can be found in the *org.eclipse.rwt* package. This class helps to load the translated .properties files into the class fields. The following snippet shows how to implement the *NLSHelper*.

```
import org.eclipse.rwt.RWT;

public class NLSHelperImpl extends NLSHelper {

  protected Object internalGetMessages(Class clazz) {
    return RWT.NLS.getISO8859_1Encoded(BUNDLE_NAME, Messages.class);
  }

}
```

Listing 32: RAP implementation for NLSHelper

As RCP was never designed to support multiple languages during runtime we cannot reuse the NLS implementation of OSGi. Therefore we need to provide our own solution as the following snippet shows.

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;

public class NLSHelperImpl extends NLSHelper {

  protected Object internalGetMessages(Class clazz) {
    ClassLoader loader = clazz.getClassLoader();
    ResourceBundle bundle = ResourceBundle.getBundle(BUNDLE_NAME, Locale
        .getDefault(), loader);
    return internalGet(bundle, clazz);
  }

  private Object internalGet(ResourceBundle bundle, Class clazz) {
    Object result;
    try {
      Constructor constructor = clazz.getDeclaredConstructor(null);
      constructor.setAccessible(true);
      result = constructor.newInstance(null);
    } catch (final Exception ex) {
      throw new IllegalStateException(ex.getMessage());
    }
    final Field[] fieldArray = clazz.getDeclaredFields();
    for (int i = 0; i < fieldArray.length; i++) {
      try {
        int modifiers = fieldArray[i].getModifiers();
        if (String.class.isAssignableFrom(fieldArray[i].getType())
```

```
            && Modifier.isPublic(modifiers)
            && !Modifier.isStatic(modifiers)) {
          try {
            String value = bundle
                .getString(fieldArray[i].getName());
            if (value != null) {
              fieldArray[i].setAccessible(true);
              fieldArray[i].set(result, value);
            }
          } catch (final MissingResourceException mre) {
            fieldArray[i].setAccessible(true);
            fieldArray[i].set(result, "");
            mre.printStackTrace();
          }
        }
      } catch (final Exception ex) {
        ex.printStackTrace();
      }
    }
    return result;
  }
}
```

Listing 33: RCP implementation of the NLSHelper

12.5 Fix your message references

Instead of accessing the static fields of the message classes we now need to reference the instance fields in our application.

```
public Object[] getElements(Object parent) {
  return new String[] { Messages.get().View_1, Messages.get().View_2, Messages.get().View_3
      };
}
```

Listing 34: Instance fields instead of static fields

With this approach we can serve multi-lingual applications with support for multiple concurrent languages - for both RCP and RAP.

12.6 Localize the plug-in manifest

Extension definitions in the plug-in manifest file can also contain strings that are subject to internationalization. In order to get this working there are a few prerequisites:

1. check out the fragment *org.eclipse.rap.equinox.registry* from the RAP CVS[6]
2. include the plug-in *org.eclipse.equinox.registry* into your workspace.

The plug-in manifest file (plugin.xml) may also contain translatable strings. Like in RCP, those strings are replaced by unique keys, prefixed with a % sign. The keys are then resolved in a *plugin.properties* file that resides in the root directory of the plug-in. For example, the internationalized version of the HelloWorld plug-in manifest file contains placeholders for the names of the view and the perspective.

```
   <extension
         point="org.eclipse.ui.views">
      <view
            id="org.eclipse.rap.helloworld.helloWorldView"
            class="org.eclipse.rap.helloworld.HelloWorldView"
            name="%helloWorldView_name">
      </view>
   </extension>

   <extension
         point="org.eclipse.ui.perspectives">
```

[6] http://www.eclipse.org/rap/cvs.php

```
      <perspective
            id="org.eclipse.rap.helloworld.perspective"
            class="org.eclipse.rap.helloworld.Perspective"
            icon="icons/icon.gif"
            name="%perspective_name">
      </perspective>
   </extension>
```

Listing 35: Sample plugin.xml with placeholders

And here's the plugin.properties:

```
helloWorldView_name = Hello World View
perspective_name = Hello World Perspective
```

Listing 36: plugin.properties

To make this work, the OSGi manifest file (MANIFEST.MF) must contain the following line:

```
Bundle-Localization: plugin
```

Listing 37: MANIFEST.MF

In summary, RAP requires special treatment to support the internationalization of extensions because of its server-side, multi-user nature. However the Equinox extension registry does the translation on startup and caches the results. In RAP, different sessions may require translations into different languages. To solve this, we had to exploit the translation mechanism of the Equinox extension registry.

To launch the application from Eclipse, you need to import the plug-in *org.eclipse.equinox.registry* as source plug-in into your workspace. Make sure that this plug-in is also included in the launch configuration (from the workspace, not from the target platform).

List of Figures

List of Tables

Listings

References

[AHL+05] Ken Arnold, David Holmes, Tim Lindholm, Frank Yellin, Frank Yellin, The Java Team, Mary Campione, Kathy Walrath, Patrick Chan, Rosanna Lee, Jonni Kanerva, James Gosling, James Gosling, James Gosling, James Gosling, Bill Joy, Bill Joy, Guy Steele, Guy Steele, Gilad Bracha, and Gilad Bracha. Java language specification, third edition, 2005.

[All03] Osgi Alliance. *OSGi Service Platform: The OSGi Alliance*. IOS Press, December 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 7 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[NW04] S. Northover and M. Wilson. *SWT: the standard widget toolkit. Volume 1*. Boston: Addison-Wesley, 2004.