# ECLIPSE ADAPTERS

A HANDS-ON, HAND-HOLDING EXPLANATION

Copyright © 2007 Jeffrey Ricker LLC
November 1, 2007
http://www.jeffreyricker.com

## Apples to oranges

When programming Eclipse plug-ins, you quickly come face to face with Eclipse adapters. If you are not familiar with the adapter pattern, adapters can be confusing. Eclipse adapters are actually very simple, and I hope to make them even simpler with this article.

Adapters work on a simple premise: Given some adaptable object A, get me the relevant object of type B for it. The Eclipse adapter interface is shown in Listing 1. The interface returns an object which is an instance of the given class associated with the object or it returns null if no such associated object can be found.

```
package org.eclipse.core.runtime;

public interface IAdaptable {
  public Object getAdapter(Class adapter);
}
```

*Listing 1 the Eclipse adapter interface*

For instance, if I wanted to go from apples to oranges then I would do something like Listing 2. In this example, `IApple` extends the `IAdaptable` interface.

```
IApple macintosh = new Macintosh();
IOrange orange = (IOrange) macintosh.getAdapter(IOrange.class);
if (orange==null)
  log("No orange");
else
 log("Created a "+ orange.getClass().getCanonicalName());
```

*Listing 2 Adapting from apples to oranges*

One of the primary uses of adapters is to separate model code from view code, as in a view-model-controller or view-model-presenter pattern. We would not want to put presentation information like icons in our apple model. We would another class to handle how to present it. We could do this with adapters as shown in Listing 3.

```
IApple apple = new Macintosh();
ILableProvider label = (ILabelProvider)apple.getAdapter(ILabelProvider.class);
String text = label.getText(apple);
```

*Listing 3 Using adapters to separate model from view*

Adapters allow us to transform objects into other purposes that the objects did not need to anticipate. For instance, the apple objects in the

Listing 3 do not need to know anything about the label provider. We can implement the `getAdapter()` method manually for each apple object, but that would defeat the purpose. Instead, we should defer the adaptation to the platform as shown in Listing 4.

```
public abstract class Fruit implements IAdaptable{

  public Object getAdapter(Class adapter){
    return Platform.getAdapterManager().getAdapter(this, adapter);
  }
}
```

*Listing 4 Adapting through the platform*

# Adapter factories

To enable the platform to manage the adaptations, you need to register one or more adapter factories with the platform. The registration can be a bit confusing, so I am going to be very specific.

```
package com.jeffricker.fruit;

import org.eclipse.core.runtime.IAdapterFactory;
import com.jeffricker.fruit.apples.IApple;
import com.jeffricker.fruit.apples.Macintosh;
import com.jeffricker.fruit.oranges.IOrange;
import com.jeffricker.fruit.oranges.Mandarin;

/**
 * Converts apples to oranges
 * @author Ricker
 */
public class OrangeAdapterFactory implements IAdapterFactory {

  public Object getAdapter(Object adaptableObject, Class adapterType) {
    if (adapterType == IOrange.class) {
      if (adaptableObject instanceof Macintosh) {
        return new Mandarin();
      }
    }
    return null;
  }

  public Class[] getAdapterList() {
    return new Class[]{ IOrange.class };
  }

}
```

*Listing 5 Apples to oranges adapter factory*

Listing 5 shows an adapter factory that converts apples to oranges. The factory enables the behavior shown earlier in Listing 2. We will detail its behavior.

- The `adaptableObject` is the object that we are starting with, the apple. The `adaptableObject` is always an object instance.
- The `adapterType` is the object to which we are adapting, the orange. The `adapterType` is always a class type, not an object instance.

Copyright © 2007 Jeffrey Ricker LLC

- The adapter list is the list of class types to which this factory can adapt objects. In this case, it is only oranges.

We must register the adapter factory with the Eclipse platform in order for it to be useful. Listing 6 shows the registration entry from the plug-in manifest file. The extension point is `org.eclipse.core.runtime.adapters`. This is where I usually mess up, so pay attention. The `adaptableType` is what we are adapting *from*. In this case, it is apples. The adapter is what we are adapting *to*. In this case, it is oranges.

```
<extension
  point="org.eclipse.core.runtime.adapters">
  <factory
    adaptableType="com.jeffricker.fruit.apples.IApple"
    class="com.jeffricker.fruit.OrangeAdapterFactory">
    <adapter
      type="com.jeffricker.fruit.oranges.IOrange">
    </adapter>
  </factory>
</extension>
```

*Listing 6 Registering the adapter factory*

There can be multiple `adapter` entries for the factory. The adapters listed in the extension point should be the same as those provided by the `getAdapterList()` method in the adapter factory.

If we look at the listings together and trace through the logic, the adapters start to make sense.

1) We create an instance of the `Macintosh` object.

```
IApple macintosh = new Macintosh();
```

2) We request the `Macintosh` to adapt to an `IOrange`

```
IOrange orange = (IOrange) macintosh.getAdapter(IOrange.class);
```

3) The `Macintosh` object forwards the request to the platform

```
public Object getAdapter(Class adapter){
  return Platform.getAdapterManager().getAdapter(this, adapter);
}
```

4) The platform finds the appropriate adapter factory through the registry.

```
<factory
  adaptableType="com.jeffricker.fruit.apples.IApple"
  class="com.jeffricker.fruit.OrangeAdapterFactory">
  <adapter type="com.jeffricker.fruit.oranges.IOrange"/>
</factory>
```

5) The platform calls the factory method, passing it an instance of the `Macintosh` object and the `IOrange` class type.

6) The adapter factory is creates an instance of the `Mandarin` object

```
public Object getAdapter(Object adaptableObject, Class adapterType) {
    if (adapterType == IOrange.class) {
      if (adaptableObject instanceof Macintosh) {
        return new Mandarin();
      }
```

Copyright © 2007 Jeffrey Ricker LLC 

```
    }
    return null;
  }
```

The confusion arises for me with the `adaptableType` parameter in the extension point. We do not have that specified in the adapter factory interface. It is buried within the logic of the factory's `getAdapter()` method. Its presence in the registry makes sense when you think about it. We ask the platform to find an adapter for a `Macintosh` object. The factories must somehow be associated to the class hierarchy of `Macintosh`. In our case the factory is registered with `IApples`. Figure 1 shows the relation between declarations in the extension point registry and the adapter factory class.

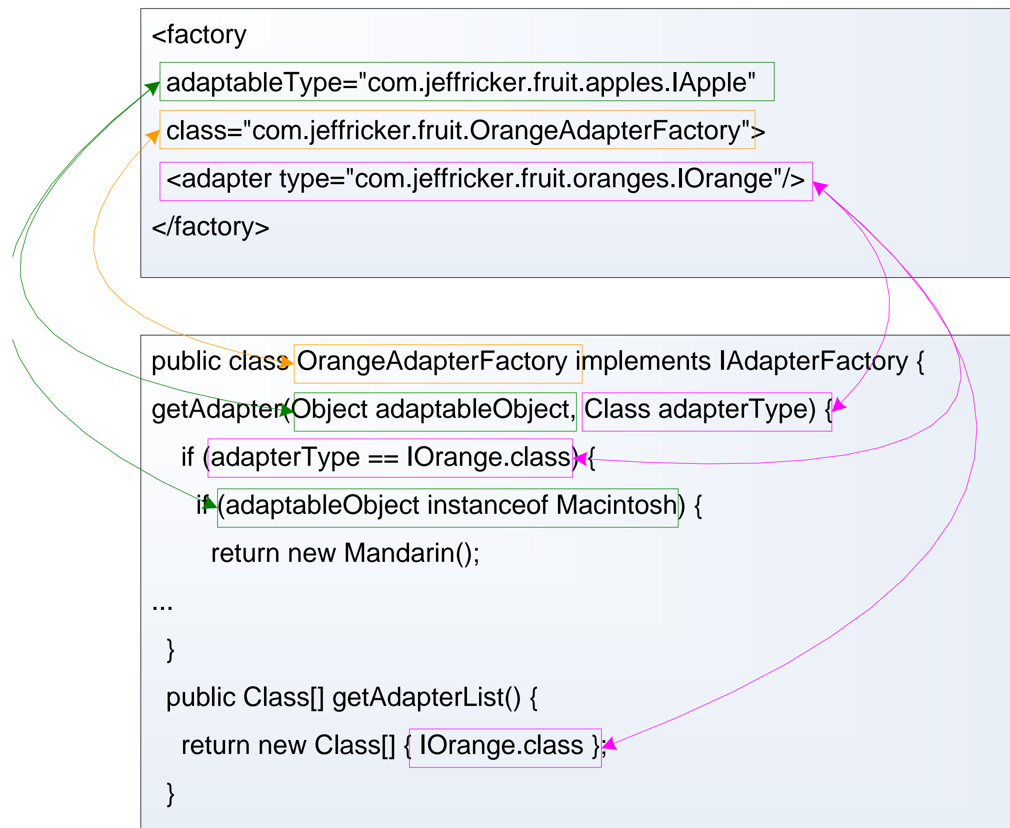


*Figure 1 Relation between factory and extension point*

# Presentation provider example

Adapting apples to oranges is a silly example of course, but I could extend the example to something more relevant. In Listing 3 I showed the adaptation of an apple object to a `ILabelProvider`, an interface used by JFace widgets for presentation. The factory for this effort is shown in Listing 7. The registration is shown in Listing 8 and sketch of the providers is shown in Listing 9.

Copyright © 2007 Jeffrey Ricker LLC

If you look at the provider classes generated by the Eclipse Modeling Framework (EMF)[1], you will see the concept of this example taken its logical conclusion.

```
package com.jeffricker.fruit.provider;

import org.eclipse.core.runtime.IAdapterFactory;
import org.eclipse.jface.viewers.IContentProvider;
import org.eclipse.jface.viewers.ILabelProvider;
import com.jeffricker.fruit.apples.IApple;
import com.jeffricker.fruit.oranges.IOrange;

public class FruitProviderAdapterFactory implements IAdapterFactory {

  private AppleProvider appleProvider;
  private OrangeProvider orangeProvider;

  /** The supported types that we can adapt to */
  private static final Class[] TYPES = {
    FruitProvider.class,  ILabelProvider.class, IContentProvider.class };

  public Object getAdapter(Object adaptableObject, Class adapterType) {
    if ((adapterType == FruitProvider.class)||
          (adapterType == ILabelProvider.class)||
          (adapterType == IContentProvider.class)){
      if (adaptableObject instanceof IApple)
        return getAppleProvider();
      if (adaptableObject instanceof IOrange)
        return getOrangeProvider();
    }
    return null;
  }

  public Class[] getAdapterList() {
    return TYPES;
  }

  protected AppleProvider getAppleProvider(){
    if (appleProvider == null)
      appleProvider = new AppleProvider();
    return appleProvider;
  }

  protected OrangeProvider getOrangeProvider(){
    if (orangeProvider == null)
      orangeProvider = new OrangeProvider();
    return orangeProvider;
  }

}
```

*Listing 7 The fruit provider factory for displaying fruit in a JFace widget*

```
<extension
  point="org.eclipse.core.runtime.adapters">
  <factory
    adaptableType="com.jeffricker.fruit.IFruit"
    class="com.jeffricker.fruit.provider.FruitProviderAdapterFactory">
    <adapter
      type="com.jeffricker.fruit.provider.FruitProvider">
```

[1] http://www.eclipse.org/emf/

```
    </adapter>
    <adapter
      type="org.eclipse.jface.viewers.IContentProvider">
    </adapter>
    <adapter
      type="org.eclipse.jface.viewers.ILabelProvider">
    </adapter>
  </factory>
</extension>
```

*Listing 8 Registering the fruit provider adapter factory*

```
package com.jeffricker.fruit.provider;

import org.eclipse.jface.viewers.IContentProvider;
import org.eclipse.jface.viewers.ILabelProvider;

public abstract class FruitProvider implements
  ILabelProvider, IContentProvider {
...
}


/**
 * Provides the display of IApple objects
 */
public class AppleProvider extends FruitProvider{
  public String getText(Object element){
    ...
  }
  public Image getIcon(Object element){
    ...
  }
}


/**
 * Provides the display of IOrange objects
 */
public class OrangeProvider extends FruitProvider {
  ...
}
```

*Listing 9 The provider classes*

# Real world example

The Eclipse Communication Framework (ECF)[2] began as a means of putting instant messaging in the Eclipse platform, but it has since expanded in scope to enable multiple means of data sharing in the Eclipse Rich Client Platform (RCP).

ECF begins with a simple interface called a *container* and uses the `IAdaptable` pattern of Eclipse to achieve specific functionality. If we were using ECF for instant messaging, then we would focus on adapting the container for presence and other interfaces.

---

[2] http://www.eclipse.org/ecf/documentation.php

Copyright © 2007 Jeffrey Ricker LLC

Listing 10 shows how to create an ECF container. The container provides a generic means for handling any type of session level protocol. Listing 11 shows how to adapt a container for managing presence, a common feature of instant messaging. The container-adapter pattern decouples the session level protocols from the services provided over those protocols.

```
// make container instance
IContainer container = ContainerFactory.getDefault()
    .createContainer("ecf.xmpp");
// make targetID
ID newID = IDFactory.getDefault()
    .createID("ecf.xmpp","slewis@ecf1.osuosl.org");
// then connect to targetID with null authentication data
container.connect(targetID,null);
```

*Listing 10 Creating an ECF connection*

```
IPresenceContainer presence = (IPresenceContainer)container
    .getAdapter(IPresenceContainer.class);
if (presence != null) {
  // The container DOES expose IPresenceContainer capabilities
} else {
  // The container does NOT expose IPresenceContainer capabilities
}
```

*Listing 11 Adapting a container for functionality*

The possibilities are sweeping. For instance, we can create our own adapter called `IMarketDataContainer` that provides streaming market data. We would create it the same way as `IPresenceContainer`. As shown in Listing 12, different market data providers might have different session level protocols, even proprietary protocols, but the container-adapter pattern would allow us to plug all of them in to our Eclipse RCP the same way.

```
IContainer container = ContainerFactory.getDefault()
    .createContainer("md.nyse");
ID newID = IDFactory.getDefault().createID("md.nyse","feed@jeffricker.com");
container.connect(targetID,null);
IMarketDataContainer marketData = (IMarketDataContainer)container
    .getAdapter(IMarketDataContainer.class);
```

*Listing 12 New container types for ECF*

The adaptor pattern is a powerful tool which you will find used throughout the Eclipse platform. I hope with the hands-on, hand holding explanation in this article that you can now unleash that power in your own RCP applications.

Copyright © 2007 Jeffrey Ricker LLC 