Refactoring for everyone

How and why to use Eclipse's automated refactoring features

David Gallardo September 09, 2003

Eclipse provides a powerful set of automated refactorings that, among other things, let you rename Java™ elements, move classes and packages, create interfaces from concrete classes, turn nested classes into top-level classes, and extract a new method from sections of code in an old method. Becoming familiar with Eclipse's refactoring tools is a good way to improve your productivity. This survey of Eclipse's refactoring features, with examples, demonstrates how and why to use each.

Why refactor?

Refactoring is changing the structure of a program without changing its functionality. Refactoring is a powerful technique, but it needs to be performed carefully. The main danger is that errors can inadvertently be introduced, especially when refactoring is done by hand. This danger leads to a common criticism of refactoring: why fix code if it isn't broken?

There are several reasons that you might want to refactor code. The first is the stuff of legend: The hoary old code base for a venerable product is inherited or otherwise mysteriously appears. The original development team has disappeared. A new version, with new features, must be created, but the code is no longer comprehensible. The new development team, working night and day, deciphers it, maps it, and after much planning and design, tears the code to shreds. Finally, painstakingly, they put it all back together according to the new vision. This is refactoring on a heroic scale and few have lived to tell this tale.

A more realistic scenario is that a new requirement is introduced to a project that requires a design change. It's immaterial whether this requirement was introduced because of an inadvertent oversight in the original plan or because an iterative approach (such as agile or test-driven development) is being used that deliberately introduces requirements throughout the development process. This is refactoring on a much smaller scale, and it generally involves altering the class hierarchy, perhaps by introducing interfaces or abstract classes, splitting classes, rearranging classes, and so on.

A final reason for refactoring, when automated refactoring tools are available, is simply as a shortcut for generating code in the first place -- something like using a spellchecker to type a word

for you when you aren't certain how it's spelled. This mundane use of refactoring -- for generating getter and setter methods, for example -- can be an effective time saver once you are familiar with the tools.

Eclipse's refactoring tools aren't intended to be used for refactoring at a heroic scale -- few tools are -- but they are invaluable for making code changes in the course of the average programmer's day, whether that involves agile development techniques or not. After all, any complicated operation that can be automated is tedium that can be avoided. Knowing what refactoring tools Eclipse makes available, and understanding their intended uses, will greatly improve your productivity.

There are two important ways that you can reduce the risk of breaking code. One way is to have a thorough set of unit tests for the code: the code should pass the tests both before and after refactoring. The second way is to use an automated tool, such as Eclipse's refactoring features, to perform this refactoring.

The combination of thorough testing and automated refactoring is especially powerful and has transformed this once mysterious art to a useful, everyday tool. This ability to change the structure of your code without altering its functionality, quickly and safely, in order to add functionality or improve its maintainability can dramatically affect the way you design and develop code, whether you incorporate it into a formal agile methodology or not.

Types of refactoring in Eclipse

Eclipse's refactoring tools can be grouped into three broad categories (and this is the order in which they appear in the Refactoring menu):

1. Changing the name and physical organization of code, including renaming fields, variables, classes, and interfaces, and moving packages and classes
2. Changing the logical organization of code at the class level, including turning anonymous classes into nested classes, turning nested classes into top-level classes, creating interfaces from concrete classes, and moving methods or fields from a class to a subclass or superclass
3. Changing the code within a class, including turning local variables into class fields, turning selected code in a method into a separate method, and generating getter and setter methods for fields

Several refactorings don't fit neatly into these three categories, particularly Change Method Signature, which is included in the third category here. Apart from this exception, the sections that follow will discuss Eclipse's refactoring tools in this order.

Physical reorganization and renaming

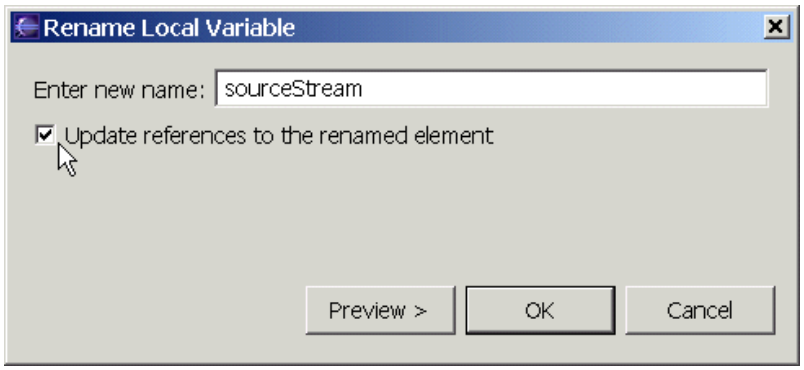
You can obviously rename or move files around in the file system without a special tool, but doing so with Java source files may require that you edit many files to update `import` or `package` statements. Similarly, you can easily rename classes, methods, and variables using a text editor's search and replace functionality, but you need to do this with care, because different classes may have like-named methods or variables; it can be tedious to go through all the files in a project making the sure that every instance is correctly identified and changed.

Eclipse's Rename and Move are able to makes these changes intelligently, throughout the entire project, without user intervention, because Eclipse understands the code semantically and is able to identify references to a specific method, variable, or class names. Making this task easy helps ensure that method, variable, and class names indicate their intent clearly.

It's quite common to find code that has inappropriate or misleading names because the code was changed to work differently than originally planned. For example, a program that looks for specific words in a file might be extended to work with Web pages by using the `URL` class to obtain an `InputStream`. If this input stream was called `file` originally, it should be changed to reflect its new more general nature, perhaps to `sourceStream`. Developers often fail to make changes like this because it can be a messy and tedious process. This, of course, makes the code confusing to the next developer who must work with it.

To rename a Java element, simply click on it in the Package Explorer view or select it in a Java source file, then select **Refactor > Rename**. In the dialog box, select the new name and choose whether Eclipse should also change references to the name. The exact fields that are displayed depend on the type of element that you select. For example, if you select a field that has getter and setter methods, you can also update the names of these methods to reflect the new field. Figure 1 shows a simple example.

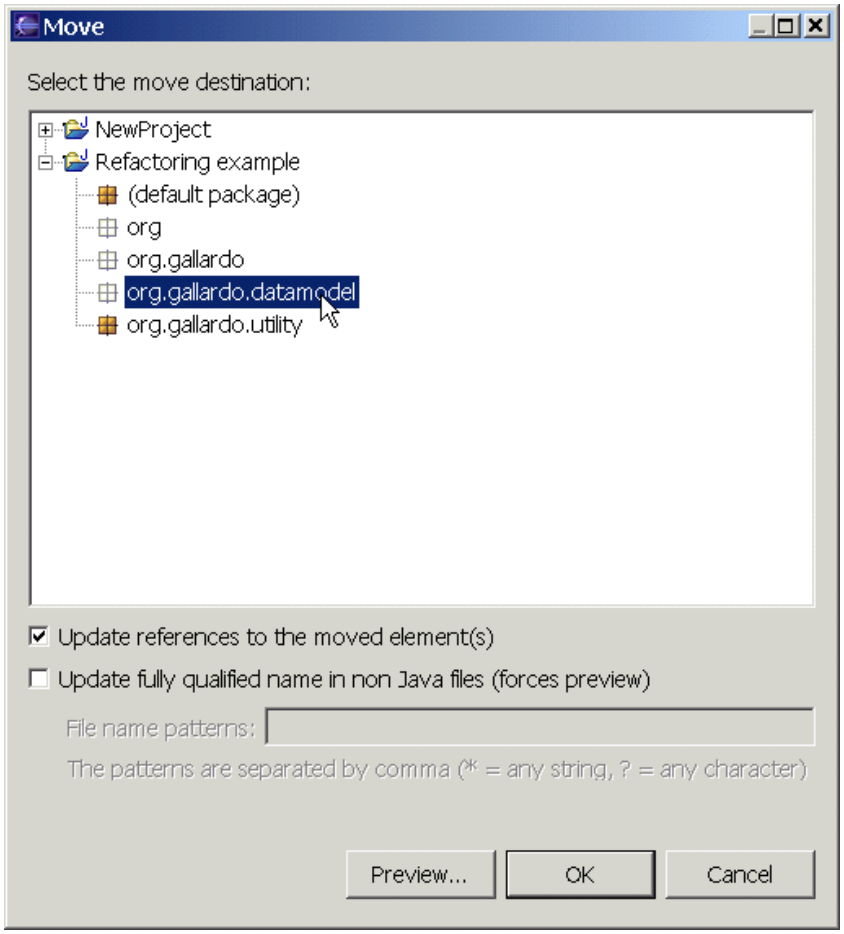
Figure 1. Renaming a local variable



Like all Eclipse refactorings, after you have specified everything necessary to perform the refactoring, you can press **Preview** to see the changes that Eclipse proposes to make, in a comparison dialog, which lets you veto or approve each change in each affected file, individually. If you have confidence in Eclipse's ability to perform the change correctly, you can instead just press **OK**. Obviously, if you are uncertain what a refactoring will do, you'll want to preview first, but this isn't usually necessary for simple refactorings like Rename and Move.

Move works very much like Rename: You select a Java element (usually a class), specify its new location, and specify whether references should also be updated. You can then choose **Preview** to examine the changes or **OK** to carry out the refactoring immediately as shown in Figure 2.

Figure 2. Moving a class from one package to another



On some platforms (notably Windows), you can also move classes from one package or folder to another by simply dragging and dropping them in the Package Explorer view. All references will be updated automatically.

Redefining class relationships

A large set of Eclipse's refactorings let you alter your class relationships automatically. These refactorings aren't as generally useful as the other types of refactorings that Eclipse has to offer, but are valuable because they perform fairly complex tasks. When they're useful, they're very useful.

Promoting anonymous and nested classes

Two refactorings, Convert Anonymous Class to Nested and Convert Nested Type to Top Level, are similar in that they move a class out of its current to scope to the enclosing scope.

An anonymous class is a kind of syntactic shorthand that lets you instantiate a class implementing an abstract class or interface where you need it, without having to explicitly give it a class name. This is commonly used when creating listeners in the user interface, for example. In Listing 1, assume that Bag is an interface defined elsewhere that declares two methods, `get()` and `set()`.

Listing 1. Bag class

```
public class BagExample
{
   void processMessage(String msg)
   {
      Bag bag = new Bag()
      {
         Object o;
         public Object get()
         {
            return o;
         }
         public void set(Object o)
         {
            this.o = o;
         }
      };
      bag.set(msg);
      MessagePipe pipe = new MessagePipe();
      pipe.send(bag);
   }
}
```

When an anonymous class becomes so large that the code becomes difficult to read, you should consider making the anonymous class a proper class; to preserve encapsulation (in other words, to hide it from outside classes that don't need to know about it), you should make this a nested class rather than a top-level class. You can do this by clicking inside the anonymous class and selecting **Refactor > Convert Anonymous Class to Nested**. Enter a name for the class, such as `BagImpl`, when prompted and then select either **Preview** or **OK**. This will change the code as shown in Listing 2.

Listing 2. Refactored Bag class

```
public class BagExample
{
   private final class BagImpl implements Bag
   {
      Object o;
      public Object get()
      {
         return o;
      }
      public void set(Object o)
      {
         this.o = o;
      }
   }

   void processMessage(String msg)
   {
     Bag bag = new BagImpl();
     bag.set(msg);
     MessagePipe pipe = new MessagePipe();
     pipe.send(bag);
   }
}
```

Convert Nested Type to Top Level is useful when you want to make a nested class available to other classes. You might, for example, be using a value object inside a class -- such as the

`BagImpl` class above. If you later decide that this data should be shared between classes, this refactoring will create a new class file from the nested class. You can do this by highlighting the class name in the source file (or clicking on the class name in the Outline view) and selecting **Refactor > Convert Nested Type to Top Level**.

This refactoring will ask you to provide a name for the enclosing instance. It may offer a suggestion, such as `example`, which you can accept. The meaning of this will be clear in a moment. After pressing **OK**, the code for the enclosing `BagExample` class will be changed as shown in Listing 3.

Listing 3. Refactored Bag class

```
public class BagExample
{
   void processMessage(String msg)
   {
      Bag bag = new BagImpl(this);
      bag.set(msg);
      MessagePipe pipe = new MessagePipe();
      pipe.send(bag);
   }
}
```

Note that when a class is nested, it has access to the outer class's members. To preserve this functionality, the refactoring will add an instance of the enclosing class **BagExample** to the formerly nested class. This is the instance variable you were previously asked to provide a name for. It also creates a constructor that sets this instance variable. The new **BagImpl** class that the refactoring creates is shown in Listing 4.

Listing 4. BagImpl class

```
final class BagImpl implements Bag
{
   private final BagExample example;
   /**
    * @paramBagExample
    */
  BagImpl(BagExample example)
   {
      this.example = example;
      // TODO Auto-generated constructor stub
   }
   Object o;
   public Object get()
   {
      return o;
   }
   public void set(Object o)
   {
      this.o = o;
   }
}
```

If you don't need to preserve access to the `BagExample` class, as is the case here, you can safely remove the instance variable and the constructor, and change the code in the `BagExample` class to the default no-arg constructor.

Moving member within the class hierarchy

Two other refactorings, Push Down and Pull Up, move class methods or fields from a class to its subclass or superclass, respectively. Suppose you have an abstract class `Vehicle`, defined as follows in Listing 5.

Listing 5. Abstract Vehicle class

```
public abstract class Vehicle
{
   protected int passengers;
   protected String motor;

   public int getPassengers()
   {
      return passengers;
   }
   public void setPassengers(int i)
   {
      passengers = i;
   }
   public String getMotor()
   {
      return motor;
   }
   public void setMotor(String string)
   {
      motor = string;
   }
}
```

You also have a subclass of `Vehicle` called `Automobile` as shown in Listing 6.

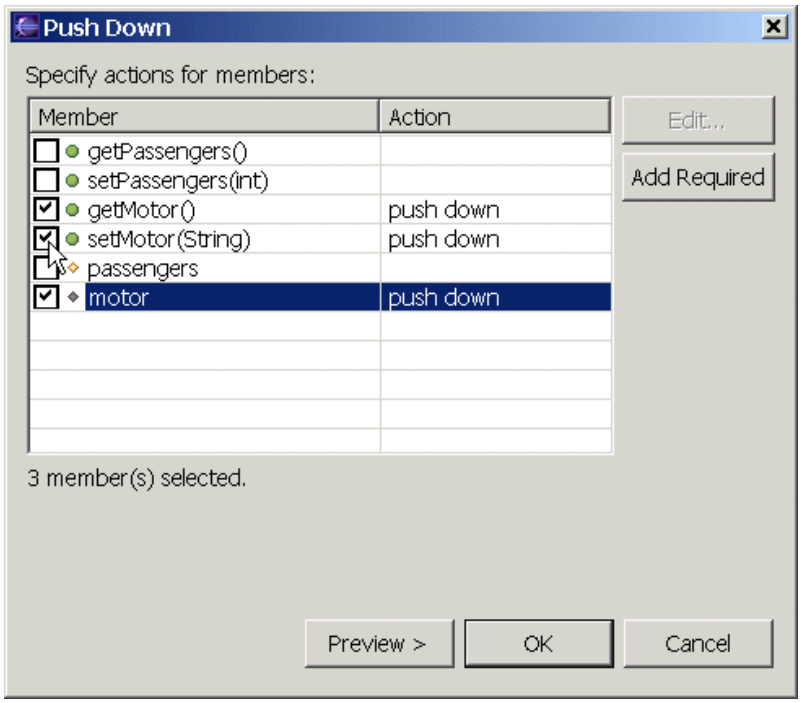
Listing 6. Automobile class

```
public class Automobile extends Vehicle
{
   private String make;
   private String model;
   public String getMake()
   {
      return make;
   }
   public String getModel()
   {
      return model;
   }
   public void setMake(String string)
   {
      make = string;
   }
   public void setModel(String string)
   {
      model = string;
   }
}
```

Notice that one of the attributes of `Vehicle` is `motor`. This is fine if you know that you will only ever deal with motorized vehicles, but if you want to allow for things like rowboats, you may want to push the `motor` attribute down from the `Vehicle` class into the Automobile class. To do this, select `motor` in the Outline view, then select **Refactor > Push Down**.

Eclipse is smart enough to realize that you can't always move a field by itself and provides a button **Add Required**, but this doesn't always work correctly in Eclipse 2.1. You need to verify that any methods that depend on this field are also pushed down. In this case there are two, the getter and setter methods that accompany the `motor` field, as shown in Figure 3.

Figure 3. Adding required members



After pressing **OK**, the `motor` field and the `getMotor()` and `setMotor()` methods will be moved to the `Automobile` class. Listing 7 shows what the `Automobile` class looks like after this refactoring.

Listing 7. Refactored Automobile class

```
public class Automobile extends Vehicle
{
   private String make;
   private String model;
   protected String motor;
   public String getMake()
   {
      return make;
   }
   public String getModel()
   {
      return model;
   }
   public void setMake(String string)
   {
      make = string;
   }
   public void setModel(String string)
   {
      model = string;
   }
   public String getMotor()
   {
```

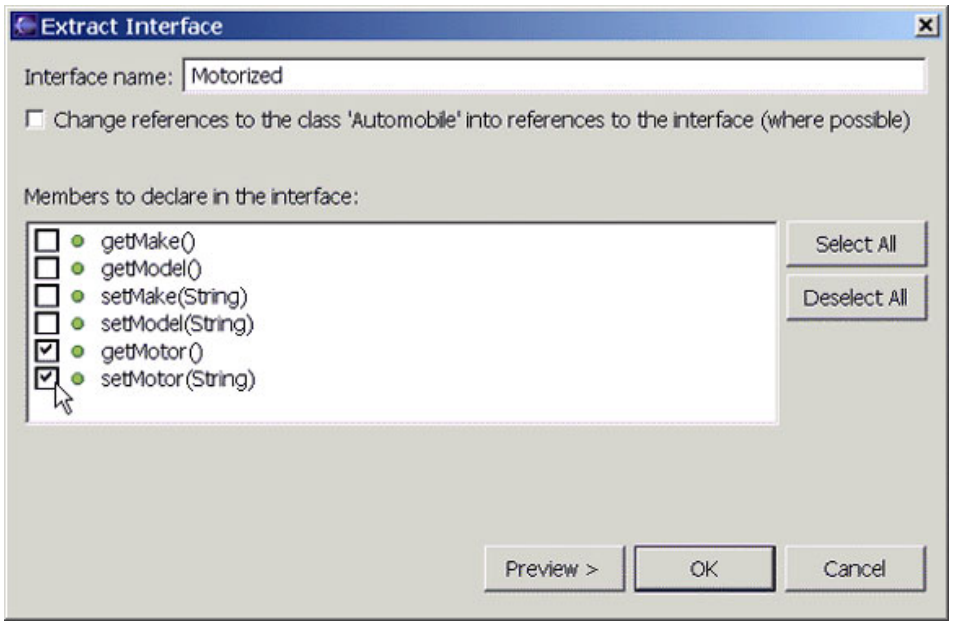
```
      return motor;
   }
   public void setMotor(String string)
   {
      motor = string;
   }
}
```

The Pull Up refactoring is nearly identical to Push down, except, of course, that it moves class members from a class to its superclass instead of subclass. You might use this if you later changed your mind and decided to move `motor` back to the `Vehicle` class. The same warning about making sure that you select all required members applies.

Having motor in the `Automobile` class means that if you create other subclasses of `Vehicle`, such as `Bus`, you'll need to add `motor` (and its associated methods) to the `Bus` class too. One way of representing relationships like this is to create an interface, `Motorized`, which `Automobile` and `Bus` would implement, but `RowBoat` would not.

The easiest way to create the `Motorized` interface is to use the Extract Interface refactoring on `Automobile`. To do this, select the `Automobile` class in the Outline view and then choose **Refactor > Extract Interface** from the menu. The dialog will allow you to select which methods you want included in the interface as shown in Figure 4.

Figure 4. Extracting the Motorized interface



After selecting **OK**, an interface is created, as shown in Listing 8.

Listing 8. Motorized interface

```
public interface Motorized
{
   public abstract String getMotor();
   public abstract void setMotor(String string);
}
```

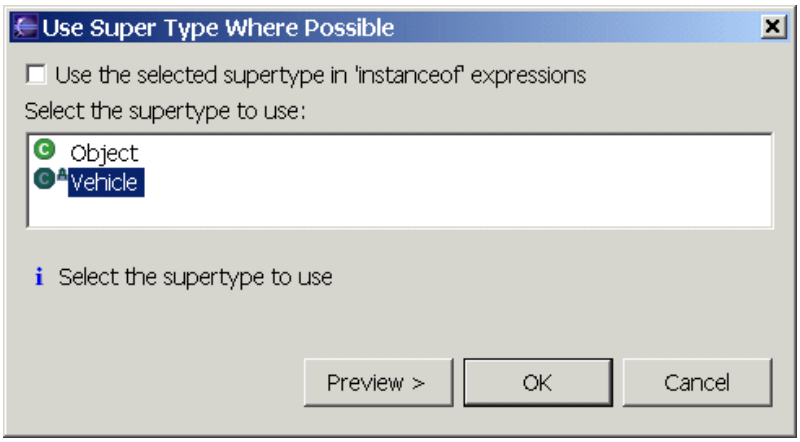
And the class declaration for `Automobile` is altered as follows:

```
public class Automobile extends Vehicle implements Motorized
```

Using a supertype

The final refactoring included in this category is Use Supertype Where Possible. Consider an application that manages an inventory of automobiles. Throughout, it uses objects of type `Automobile`. If you wanted to be able to handle all types of vehicles, you could use this refactoring to change references to `Automobile` to references to Vehicle (see Figure 5). If you perform any type-checking in your code using the `instanceof` operator, you will need to determine whether it is appropriate to use the specific type or the supertype and check the first option, Use the selected supertype in 'instanceof' expressions, appropriately.

Figure 5. Changing Automobile to its supertype, Vehicle



The need for using a supertype arises frequently in the Java language, especially when the Factory Method pattern is used. Typically this is implemented by having an abstract class that has a static `create()` method that returns a concrete object implementing the abstract class. This can be useful if the type of concrete object that must be created depends on implementation details that are of no interest to client classes.

Changing code within a class

The largest variety of refactorings are those that reorganize code within a class. Among other things, these allow you to introduce (or remove) intermediate variables, create a new method from a portion of an old one, and create getter and setter methods for a field.

Extracting and inlining

There are several refactorings beginning with the word Extract: Extract Method, Extract Local Variable, and Extract Constants. The first one, Extract Method, as you might expect, will create a new method from code you've selected. Take, for example, the `main()` method in the class in Listing 8. It evaluates command-line options and if it finds any that begin with `-D`, stores them as name-value pairs in a `Properties` object.

Listing 8. main()

```
import java.util.Properties;
import java.util.StringTokenizer;
public class StartApp
{
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i= 0; i < args.length; i++)
      {
         if(args[i].startsWith("-D"))
         {
           String s = args[i].substring(2);
           StringTokenizer st = new StringTokenizer(s, "=");
            if(st.countTokens() == 2)
            {
              props.setProperty(st.nextToken(), st.nextToken());
            }
         }
      }
      //continue...
   }
}
```

There are two main cases where you might want to take some code out of a method and put it in another method. The first case is if the method is too long and does two or more logically distinct operations. (We don't know what else this `main()` method does, but from the evidence we see here, that's not a reason for extracting a method here.) The second case is if there is a logically distinct section of code that can be re-used by other methods. Sometimes, for example, you find yourself repeating several lines of code in several different methods. That's a possibility in this case, but you probably wouldn't perform this refactoring until you actually needed to re-use this code.

Assuming there is another place where you need to parse name-value pairs and add them to a `Properties` object, you could extract the section of code that includes the `StringTokenizer` declaration and following `if` clause. To do this, highlight this code and then select **Refactor > Extract Method** from the menu. You'll be prompted for a method name; enter `addProperty`, and then verify that the method has two parameters, `Properties prop` and `Strings`. Listing 9 shows the class after Eclipse extracts the method `addProp()`.

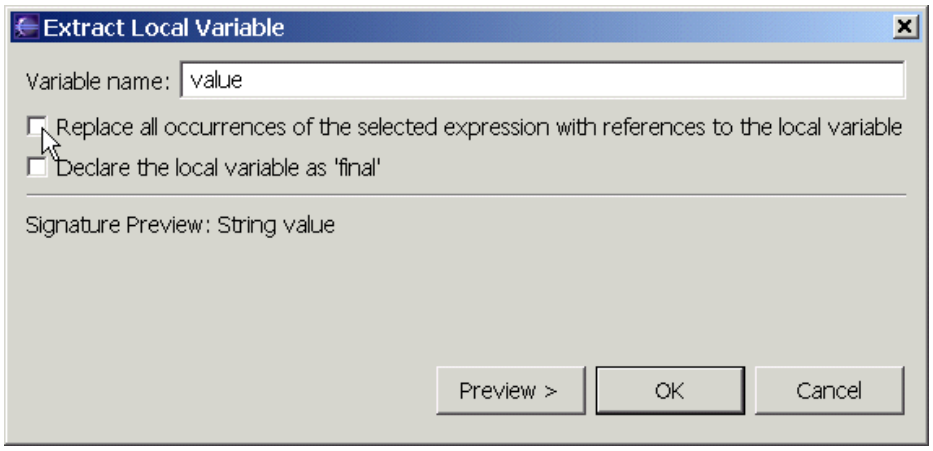
Listing 9. addProp() extracted

```
import java.util.Properties;
import java.util.StringTokenizer;
public class Extract
{
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i = 0; i < args.length; i++)
      {
         if (args[i].startsWith("-D"))
         {
            String s = args[i].substring(2);
            addProp(props, s);
         }
      }
```

```
   }
   private static void addProp(Properties props, String s)
   {
      StringTokenizer st = new StringTokenizer(s, "=");
      if (st.countTokens() == 2)
      {
         props.setProperty(st.nextToken(), st.nextToken());
      }
   }
}
```

The Extract Local Variable refactoring takes an expression that is being used directly and assigns it to a local variable first. This variable is then used where the expression used to be. For example, in the `addProp()` method above, you can highlight the first call to `st.nextToken()` and select **Refactor > Extract Local Variable**. You will be prompted to provide a variable; enter `key`. Notice that there is an option to replace all occurrences of the selected expression with references to the new variable. This is often appropriate, but not in this case of the `nextToken()` method, which (obviously) returns a different value each time it is called. Make sure this option is not selected; see Figure 6.

Figure 6. Don't replace all occurrences of selected expression



Next, repeat this refactoring for the second call to `st.nextToken()`, this time calling the new local variable `value`. Listing 10 shows the code after these two refactorings.

Listing 10. Refactored code

```
private static void addProp(Properties props, String s)
   {
     StringTokenizer st = new StringTokenizer(s, "=");
      if(st.countTokens() == 2)
      {
         String key = st.nextToken();
         String value = st.nextToken();
        props.setProperty(key, value);
      }
   }
```

Introducing variables in this way provides several benefits. First, by providing meaningful names to the expressions, it makes explicit what the code is doing. Second, it makes it easier to debug

the code, because we can easily inspect the values that the expressions return. Finally, in cases where multiple instances of an expression can be replaced with a single variable, this can be more efficient.

Extract Constant is similar to Extract Local Variable, but you must select a static, constant expression, which the refactoring will convert to a static final constant. This is useful for removing hard-coded numbers and strings from your code. For example, in the code above we used `-D"` for the command line option defining a name-value pair. Highlight `-D"` in the code, select **Refactor > Extract Constant**, and enter `DEFINE` as the name of the constant. This refactoring will change the code as shown in Listing 11.

Listing 11. Refactored code

```
public class Extract
{
   private static final String DEFINE = "-D";
   public static void main(String[] args)
   {
      Properties props = new Properties();
      for (int i = 0; i < args.length; i++)
      {
         if (args[i].startsWith(DEFINE))
         {
            String s = args[i].substring(2);
            addProp(props, s);
         }
      }
   }
   // ...
```

For each Extract... refactoring, there is a corresponding Inline... refactoring that performs the reverse operation. For example, if you highlight the variable s in the code above, select **Refactor > Inline...**, then press **OK**, Eclipse use the expression `args[i].substring(2)` directly in the call to `addProp()` as follows:

```
        if(args[i].startsWith(DEFINE))
         {
            addProp(props,args[i].substring(2));
         }
```

This can be marginally more efficient than using a temporary variable and, by making the code terser, makes it either easier to read or more cryptic, depending on your point of view. Generally, however, inlining like this does not have much to recommend it.

In the same way that you can replace a variable with an inline expression, you can also highlight a method name or a static final constant. Select **Refactor > Inline...** from the menu, and Eclipse will replace method calls with the method code, or references to the constant with the constants value, respectively.

Encapsulating fields

It's generally not considered good practice to expose the internal structure of your objects. That's why the `Vehicle` class, and its subclasses, have either private or protected fields, and public setter

and getter methods to provide access. These methods can be generated automatically in two different ways.

One way to generate these methods is to use the **Source > Generate Getter and Setter**. This will display a dialog box with the proposed getter and setter methods for each field that does not already have one. This is not a refactoring, though, because it does not update references to the fields to use the new methods; you'll need to that yourself if necessary. This option is a great time saver, but it's best used when creating a class initially, or when adding new fields to a class, because no other code references these fields yet, so there's no other code to change.

The second way to generate getter and setter methods is to select the field and then choose **Refactor > Encapsulate Field** from the menu. This method only generates getters and setters for a single field at a time, but in contrast to **Source > Generate Getter and Setter**, it also changes references to the field into calls to the new methods.

For example, start fresh with a new, simple version of the `Automobile` class, as shown in Listing 12.

Listing 12. Simple Automobile class

```
public class Automobile extends Vehicle
{
   public String make;
   public String model;
}
```

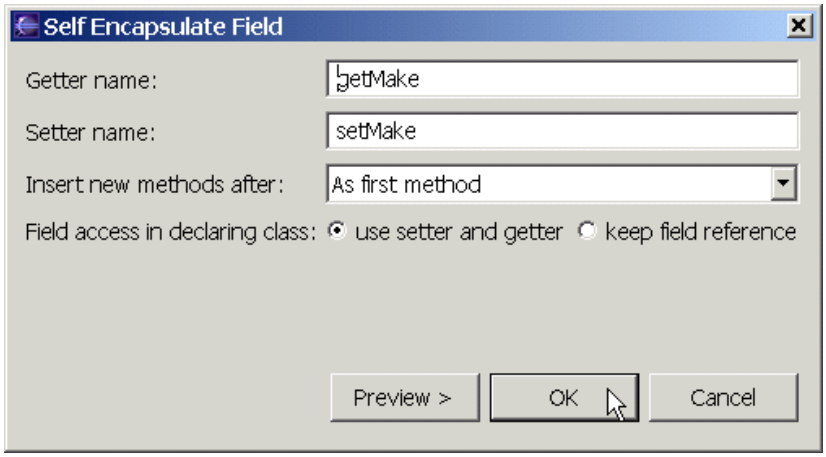
Next, create a class that instantiates `Automobile` and accesses the `make` field directly, as shown in Listing 13.

Listing 13. Instantiate Automobile

```
public class AutomobileTest
{
   public void race()
   {
      Automobilecar1 = new Automobile();
      car1.make= "Austin Healy";
      car1.model= "Sprite";
      // ...
   }
}
```

Now encapsulate the `make` field by highlighting the field name and selecting **Refactor > Encapsulate Field**. In the dialog, enter names for the getter and setter methods -- as you might expect, these are `getMake()` and `setMake()` by default. You can also choose whether methods that are in the same class as the field will continue to access the field directly or whether these references will be changed to use the access methods like all other classes. (Some people have a strong preference one way or the other, but as it happens, it doesn't matter what you choose in this case, since there are no references to the `make` field in `Automobile`). See Figure 7.

Figure 7. Encapsulating a field



After pressing **OK**, the `make` field in the **Automobile** class will be private and will have `getMake()` and `setMake()` methods as shown in Listing 14.

Listing 14. Refactored Automobile class

```
public class Automobile extends Vehicle
{
   private String make;
   public String model;

   public void setMake(String make)
   {
      this.make = make;
   }

   public String getMake()
   {
      return make;
   }
}
```

The `AutomobileTest` class will also be updated to use the new access methods, as shown in Listing 15.

Listing 15. AutomobileTest class

```
public class AutomobileTest
{
   public void race()
   {
      Automobilecar1 = new Automobile();
      car1.setMake("Austin Healy");
      car1.model= "Sprite";
      // ...
   }
}
```

Change Method Signature

The final refactoring considered here is the most difficult to use: Change Method Signature. It's fairly obvious what this does -- change the parameters, visibility, and return type of a method. What isn't so obvious is the effect these changes have on the method or on the code that calls the

method. There is no magic here. If the changes cause problems in the method being refactored -- because it leaves undefined variables or mismatched types -- the refactoring operations will flag these. You have the option to accept the refactoring anyway and correct the problems afterwards, or cancel the refactoring. If the refactoring causes problems in other methods, these are ignored and you must fix them yourself after the refactoring.

To clarify this, consider the following class and method in Listing 16.

Listing 16. MethodSigExample class

```
public class MethodSigExample
{
   public int test(String s, int i)
   {
      int x = i + s.length();
      return x;
   }
}
```

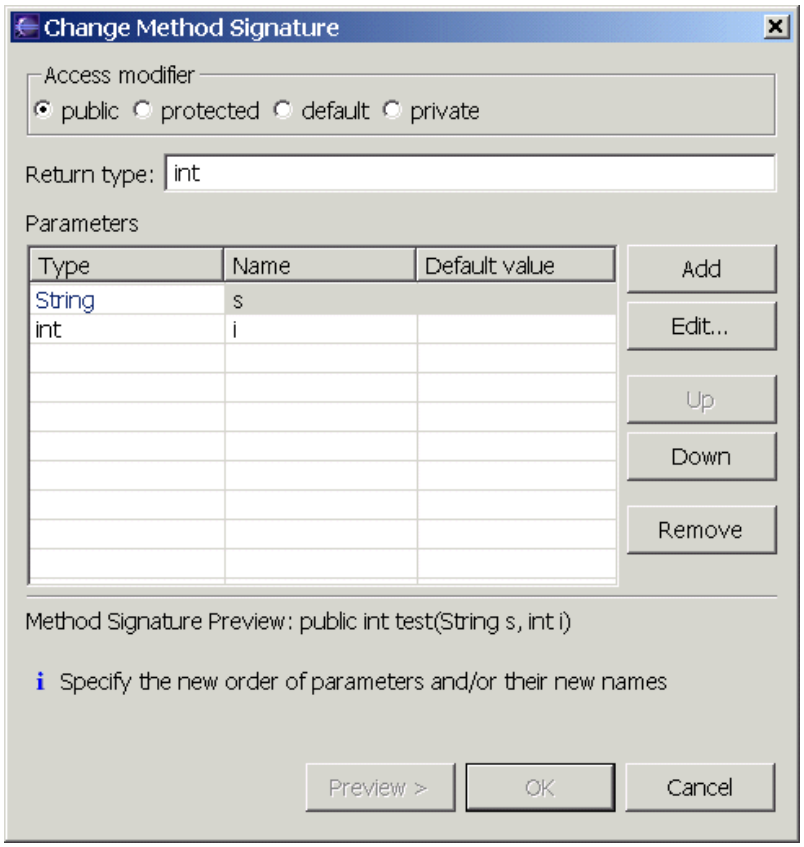
The method `test()` in the class above is called by a method in another class, as shown in Listing 17.

Listing 17. callTest method

```
public void callTest()
   {
     MethodSigExample eg = new MethodSigExample();
     int r = eg.test("hello", 10);
   }
```

Highlight `test` in the first class and select **Refactor > Change Method Signature**. The dialog box in Figure 8 will appear.

Figure 8. Change Method Signature options



The first option is to change the method's visibility. In this example, changing it to protected or private would prevent the `callTest()` method in the second class from accessing. (If they were in separate packages, changing access to default would also cause this problem.) Eclipse will not flag this error while performing the refactoring; it's up to you to select an appropriate value.

The next option is to change the return type. Changing the return type to `float`, for example, isn't flagged as an error because an `int` in the `test()` method's return statement is automatically promoted to `float`. Nonetheless, this will result in a problem in the `callTest()` in the second class, because a `float` cannot be converted to `int`. You will need to either cast the return value returned by `test()` to `int` or change the type of `r` in `callTest()` to `float`.

Similar considerations apply if we change the type of the first parameter from `String` to `int`. This will be flagged during refactoring because it causes a problem in the method being refactored: `int` does not have a `length()` method. Changing it to `StringBuffer`, however, will not be flagged as problem, because it does have a `length()` method. This will, of course, cause a problem in the `callTest()` method, because it is still passing a `String` when it calls `test()`.

As mentioned previously, in cases where this refactoring results in an error, whether flagged or not, you can continue by simply correcting the errors on a case-by-case basis. Another approach is to preempt errors. If you want to remove the parameter `i`, because it's unneeded, you could start by removing references to it in the method being refactored. Removing the parameter will then go more smoothly.

One final thing to be explained is the Default Value option. This is only used when a parameter is being added to the method signature. It is used to provide a value when the parameter is added to callers. For example, if we add a parameter of type `String`, with a name `n`, and a default value of `world`, the call to `test()` in the `callTest()` method will be changed as follows:

```
public void callTest()
{
   MethodSigExample eg = new MethodSigExample();
   int r = eg.test("hello", 10, "world");
}
```

The point to take away from this seemingly dire discussion about the Change Method Signature refactoring is not that it is problematic, but rather that it is a powerful, time-saving refactoring that often requires thoughtful planning to be used successfully.

Summary

Eclipse's tools make refactoring easy, and becoming familiar with them can help you improve your productivity. Agile development methods, which add program features iteratively, depend on refactoring as a technique to alter and extend a program's design. But even if you are not using a formal method that requires refactoring, Eclipse's refactoring tools provide a time-saving way to make common types of code changes. Taking some time to become familiar with them so that you can recognize the situations where they can be applied is a worthwhile investment of your time.

Related topics

- The key text on refactoring is *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts (Addison-Wesley, 1999).
- Refactoring, as an ongoing process, is discussed by the author in the context of designing and developing a project in Eclipse in *Eclipse In Action: A Guide for Java Developers*, by David Gallardo, Ed Burnette, and Robert McGovern (Manning, 2003).
- Patterns (such as the Factory Method mentioned in this article) are an important tool for understanding and discussing object-oriented design. The classic text is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley, 1995).
- One drawback for the Java programmer is that the examples in *Design Patterns* use C++; for a book that translates patterns to the Java language, see *Patterns in Java, Volume One: A Catalog of Reusable Design Patterns Illustrated with UML*, Mark Grand (Wiley, 1998).
- For an introduction to one variety of agile programming, see *Extreme Programming Explained: Embrace Change*, by Kent Beck (Addison-Wesley, 1999).
- Martin Fowler's Web site is refactoring central on the Web.
- For more information on unit testing with JUnit, visit the JUnit Web site.
- "Java design patterns 101" is David's introductory tutorial on patterns (*developerWorks*, January 2002).
- In "Getting started with the Eclipse Platform," David provides a starting point for learning more about Eclipse (*developerWorks*, November 2002).
- Find more articles for Eclipse users in the Open source projects zone on *developerWorks*. Also see the latest Eclipse technology downloads on *alphaWorks*.