A Short Introduction to Refactoring

Carl Carenvall and Tobias Wrigstad

All software rots. A perfectly good design choice six months ago might turn into an almost obvious stupidity due to subsequent changes to the code that were impossible to predict. Perhaps the variable `car` nowadays also references bicycles, and this should be reflected with a name change? Or all checks in client code that for what type of vehicle is in the `car` variable should be replaced with a single polymorphic method call that works regardless of the vehicle type? Or maybe 20 “quick fixes” has made the code smell nasty in many places, and it is time to clean the mess up before it becomes positively toxic.

It is inherent in programming that everyone’s code needs constant refactoring to stay in shape.

Refactoring is a term coined by Martin Fowler, a well-known software architect. It involves changing code one small step at a time. The steps are intended to be *semantics-preserving*, that is, they are not supposed to change the program’s behaviour. Just the way the behaviour is implemented. For example, moving a field closer to where it is more commonly accessed, breaking repetitive blocks that almost do the same thing out into a single more general function to allow them to be updated and testing in one single place.

Moving things around in the code, however, is not without danger. “If it ain’t broke, why fix it?” someone might say, especially if a deadline is coming up. Luckily, most modern programmer IDE’s support many common refactoring operations, like renaming a variable or a class and having all the uses of the renamed name in the code be automatically updated. Using tools that “guarantee” correctness of program transformations is safer and less prone to errors than going manual.

Furthermore, refactoring is best aided by a large set of tests that can be run after every transformation to make sure the program hasn’t broken following the change. Thus, the workflow of refactoring is generally:

1. Detect a “bad smell” in the code
2. Figure out a good cause of action to get rid of it
3. Running the program’s tests to see that they work
4. Performing the changes using suitable tools, like the refactoring tools in Eclipse, to the largest extent possible
5. After each completed step, running the tests making sure they still work

Many refactoring changes are pretty straight forward, so lets just get started with an example of making sure classes are in the right places (in the right packages). This is a simple refactoring pattern called “Move class” and is supported directly in Eclipse[1].

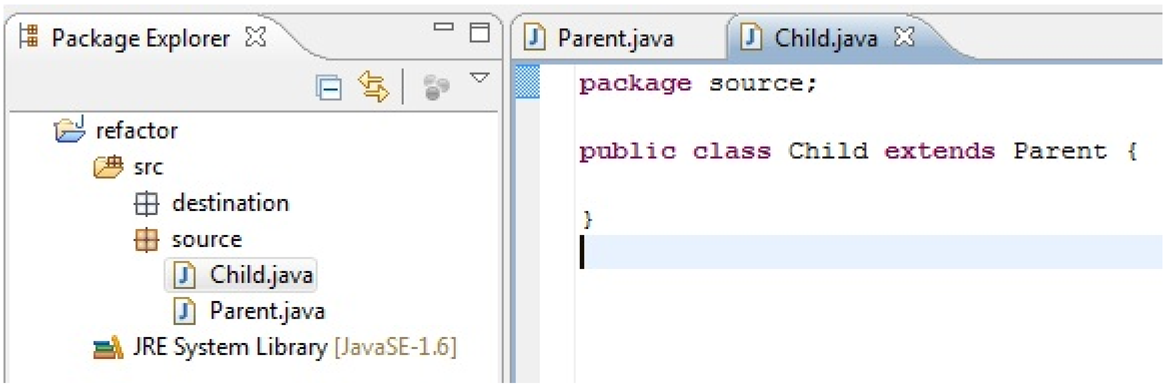
Depending on your system, there are two ways to go about making this change. The first is by selecting one or more targets in the **Package Explorer** (typically to the left), then either right click and select `refactor` → `move` in the contextual menu or select `refactor` → `move` in the menu bar at the top. This will lead to a window where you select a target, and after that a settings window for the move (described further down).

Some systems allow a slightly easier way though: just select one or more files in the **Package Explorer**, then drag and drop them to the desired package. This will effectively work as both selecting `refactor` → `move` (what else could it mean?), as well as selection of target.

[1]There are many standard “patterns” representing changes that programmes frequently make in their code bases. Renaming and moving a two examples of common patterns.

A Simple Example

Lets say we have something looking like this

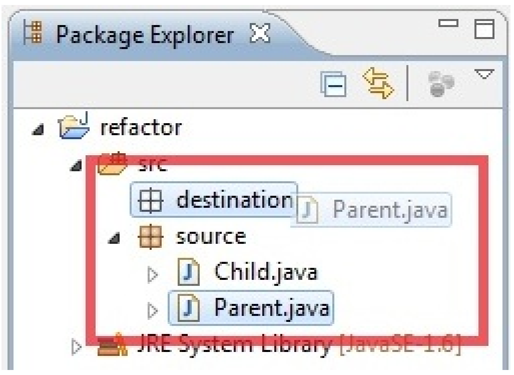


```java
package source;

public class Child extends Parent {

}
```

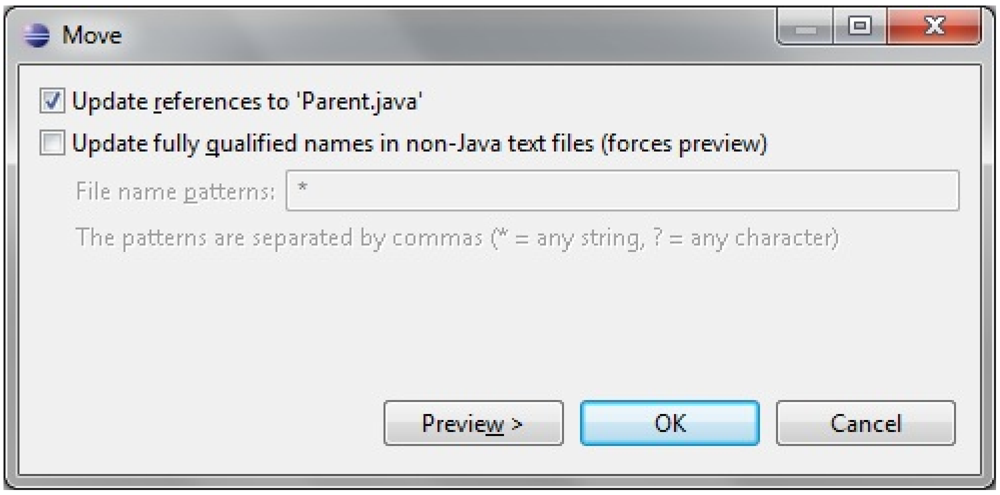
And we want to move the class Parent from package source to package destination.

Note that the class Child inherits from the class Parent.

Just drag and drop Parent.java to the desired package (in this case called destination)



As you drop it, you should be met by this



Move

[✓] Update references to 'Parent.java'

[ ] Update fully qualified names in non-Java text files (forces preview)

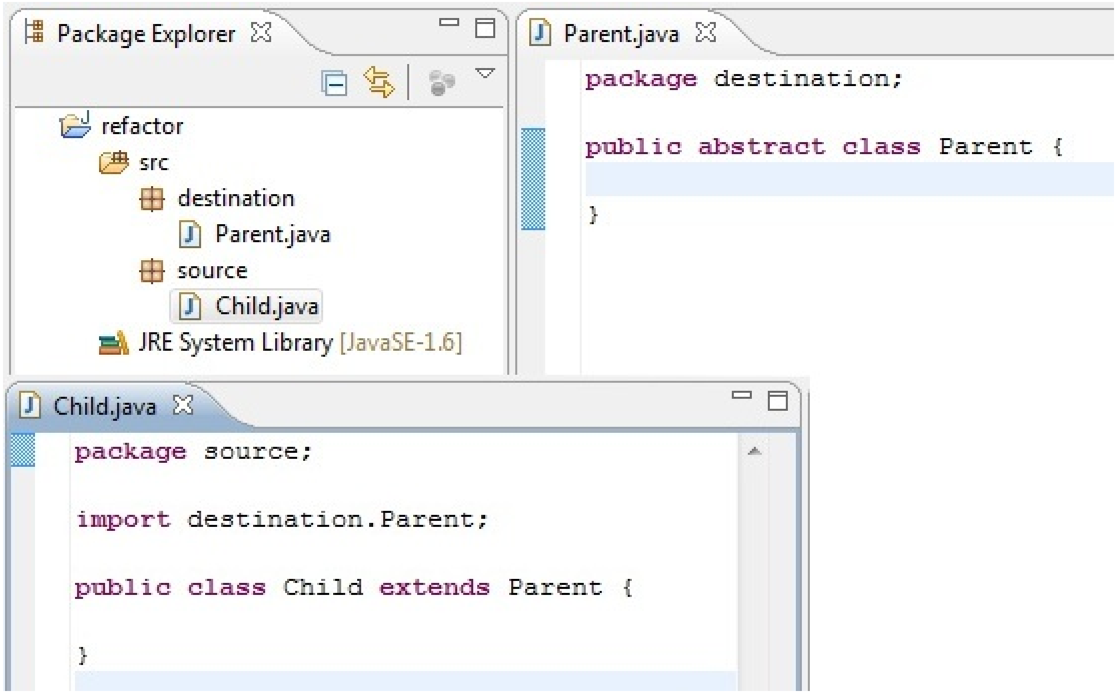
File name patterns: *

The patterns are separated by commas (* = any string, ? = any character)

The options here are pretty straight forward.

You will almost certainly want "Update references" to the class you are moving. You can also have a look at the results of the move before it is made with the preview button.

After the move we are left with something like this



```
package destination;

public abstract class Parent {

}
```

```
package source;

import destination.Parent;

public class Child extends Parent {

}
```

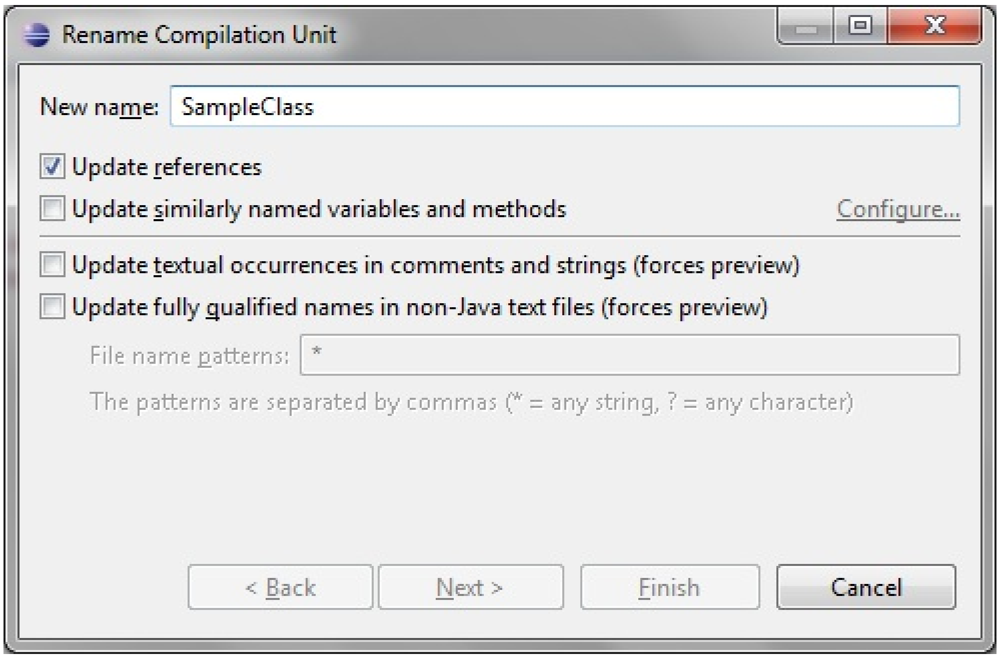
Note that not only has the package name in `Parent.java` been changed along with its location, but Child.java now imports the package needed to use `Parent.java`.

Renaming

We have now seen how a change that propagated into unrelated files was greatly facilitated by the Eclipse IDE (similar functionality can be found in many other tools as well, like Netbeans, or stand-alone refactoring tools). Let's move on to our next example, which is that of simple renaming. In this case, we want to rename a class' name to better reflect how it is being used in the program. Similar stories happen with variables, method names, etc. all the time too.

Again, the refactoring menu comes to our aid, both in the menu bar at the top and in the contextual menu when you right-click the class name. Renaming for example a variable is just as easy as renaming a class; just mark it and the choose rename. When the new name is entered all instances of the variable or class will be changed.

By selecting a file in the package explorer and choosing to rename it you get access to some more advanced options.



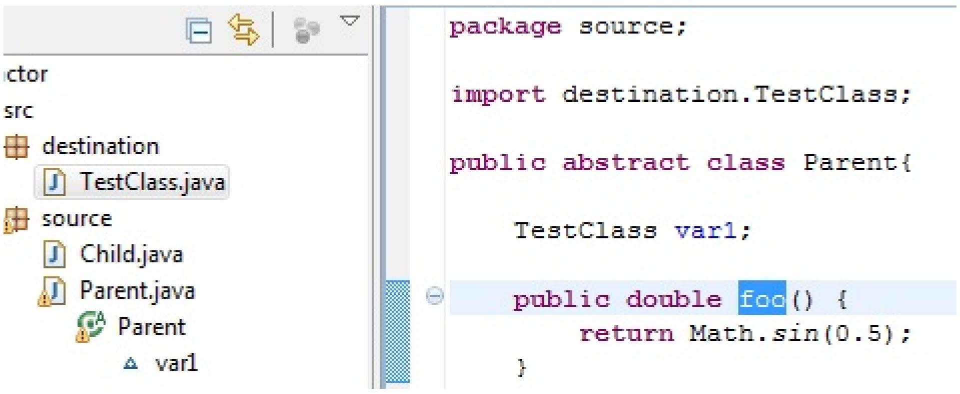
New name: SampleClass

[✓] Update references
[ ] Update similarly named variables and methods
[ ] Update textual occurrences in comments and strings (forces preview)
[ ] Update fully qualified names in non-Java text files (forces preview)

File name patterns: *

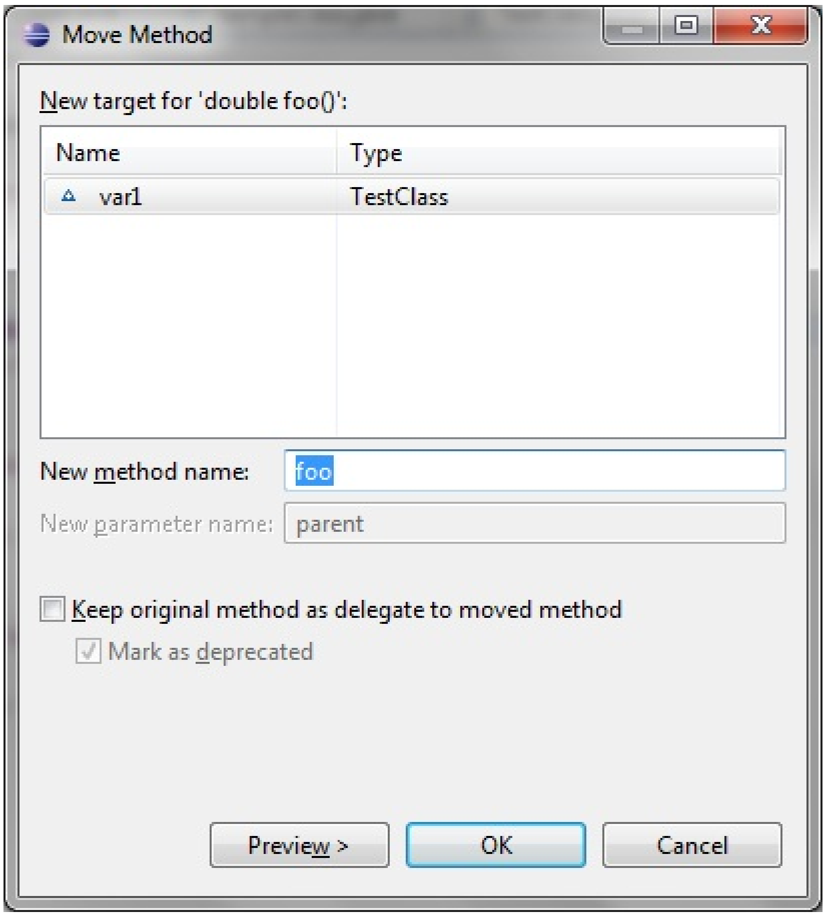
The patterns are separated by commas (* = any string, ? = any character)

Let's go ahead and do the same thing with a method name. This turns out to be almost as easy as moving a class between packages. First of all, there must be a reference in the class from which the method is being moved to the destination class. Otherwise, the tool doesn't know where to select the possible target classes from.



```
package source;

import destination.TestClass;

public abstract class Parent{

    TestClass var1;

    public double foo() {
        return Math.sin(0.5);
    }
```

Once you have a dependency (like an import or other mentioning, add if you need it), you just mark the method to be moved, right-click (or select from the top menu bar) `refactor` → `move`. You will then be met with a window looking something like this



New target for 'double foo()':

Name	Type
var1	TestClass

New method name: foo

New parameter name: parent

☐ Keep original method as delegate to moved method

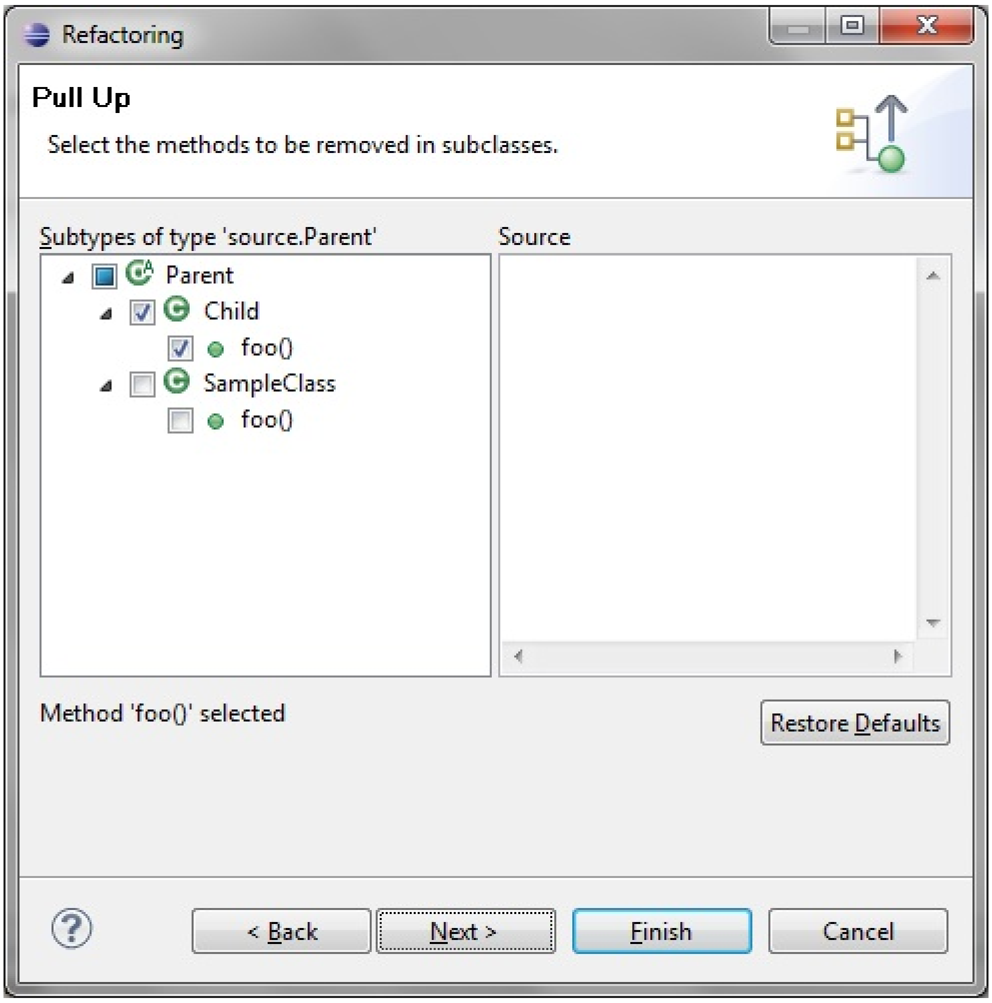
☑ Mark as deprecated

Simply select your destination, and the method will be moved to the chosen class.

As you get going on this you notice there are even specific functionality for moving methods and variable within the class hierarchy.

The commands for this are `refactor` → `push` to move things down in the class hierarchy (to the children), and `refactor` → `pull` to move things up in the class hierarchy (to the parents).

When you try pull you notice that if you press next you get the following window



Here you can select which classes (subclasses to the class you are pulling the method to) the old method should be removed from. Very useful if you want some subclasses to keep their specific implementations.

Notice that you can go into preview in order to select which classes should receive the selected members and which should not.

End Notes

Eclipse supports many other refactoring patterns, such as the ability to extract a class or an interface class from an existing class and extracting variables. The point of refactoring is to preserve the code's behaviour while making changes. If a certain refactoring should for some reason blow something up, it can be changed back with undo!

It is not a bad idea to play around with the refactoring tools to get a feeling for them. They will serve you well in the future, not only on this course.