DZone Refcardz

CONTENTS INCLUDE:

Refactoring Patterns

By Kevin Rutherford

INTRODUCTION

Refactoring has been described as "the art of safely improving the design of existing code" (Martin Fowler, see refs). Refactoring is thus a process of software source code transformation.

Refactoring does not involve adding new features. Refactoring should be performed when the code is working and all of its tests are passing. And it should be performed before adding the next feature. Thus, refactoring is intended to find the "best" implementation of the features currently present in the code.

Refactoring also does not involve rewriting or replacing large chunks of code. Refactoring is a gradual, evolutionary process, intended to "preserve the knowledge embedded in the existing code."

Refactoring is the vehicle for emergent design. Knowing how and when to refactor means that your code doesn't need to be completely designed up front. Refactoring makes it safe to change your design later, and indeed makes it more cost-effective to do so.

Refactoring stops when the code's design is as "simple" as we can cost-effectively make it for now. Four criteria for defining "simple" are the following rules from Extreme Programming:

1. Passes all of the tests.
2. Communicates the programmer's intentions, i.e., has good names for every important concept.
3. Expresses everything once and only once, i.e., it duplicates no code or logic or knowledge.
4. Includes nothing that's unnecessary.

These "rules" are subjective, and are listed above in order of priority. But they do provide a good rule of thumb in helping to decide what to refactor, and whether further refactoring is needed.

WHAT IS A REFACTORING?

At the smallest scale, a "refactoring" is a minimal, safe transformation of your code that keeps its behavior unchanged.

Each refactoring has an inverse. For example, ExtractMethod, which moves a code fragment into a new method and replaces the original with a call to that method, is the inverse of InlineMethod. Neither is "better" or "worse" than the other. Rather, each is appropriate under certain circumstances.

Each refactoring will comprise a series of steps, some of which may temporarily break the software. The refactoring is only complete when correct behavior has been restored by completion of all of its steps.

Some refactorings have multiple alternative "recipes" (sets of safe steps that achieve the refactoring's goal). The most important aspect of carrying out a refactoring is to find a recipe that minimizes the amount of time during which the code is broken.

Most minimal, safe refactorings have the same basic outline:

1. Check that the refactoring is safe to perform, and that its pre-conditions are met.
2. Introduce a new code element.
3. Locate all existing code elements that need to be migrated to use the new element.
4. For each client to be migrated:
   - Migrate the client of the old code so that it uses the new code instead.
   - Run the tests.
5. Optionally, save a checkpoint.
6. Delete the old code that is now unused.
7. Run the tests.
8. Save a checkpoint.

REFACTORING EXAMPLES

There are dozens of well-defined refactoring recipes for a huge range of small, specific tasks. Several of the books in the Further Reading at the end of this document contain catalogs of these refactorings, and in most cases they give step-by-step mechanics for carrying out the refactoring. Here we provide a taster by looking in detail at four specific and quite varied refactorings.

Inline Temporary Variable

Replace all uses of a local variable by inserting copies of its assigned value.

We begin with a simple method:

```
public double applyDiscount() {
  double basePrice = _cart.totalPrice();
  return basePrice * discount();
}
```

Step 1: Safety Check.

First, we check that the refactoring's pre-conditions are met. We can do that in this case by making the temporary variable final:

```
public double applyDiscount() {
  final double basePrice = _cart.totalPrice();
  return basePrice * discount();
}
```

Compiling now will tell us whether the refactoring is safe to perform. Leaning on the compiler like this is a trick that is useful in many refactorings (although it only works in compiled languages).

We can also test at this point and even commit if we think we may be distracted away from the task in the next few seconds. We are at a safe "base" (and, arguably, we have already improved the code).

Step 2: Migrate each client.

Next, we identify all places in the code that need to change. In this case there's only one, in the return statement. We change it to use the value that's currently assigned to the temp:

```
public double applyDiscount() {
  final double basePrice = _cart.totalPrice();
  return _cart.totalPrice() * discount();
}
```

Compile and test (and commit if you're nervous). It's important to get into the habit of running the tests frequently because it significantly enhances one's feeling of safety. Most programming errors are created under conditions of stress, and simple practices such as this can save hours of debugging later.

It is thus also important to have tests that run quickly. In cases such as this, we would probably only run the tests for the current class, so at least these must be fast.

(Note that the tests could fail here if _cart.totalPrice() has side effects, because we are now calling it twice. If that happens, they have saved us from making an unsafe refactoring, so we back out the change and walk away.)

Step 3: Remove the old code.

Finally, we can remove the now-obsolete declaration:

```
public double applyDiscount() {
  return _cart.totalPrice() * discount();
}
```

Compile and test, and definitely commit this version to source control.

Most of the available refactoring tools provide an automated implementation of this refactoring.

Extract method

Create a new method by extracting a code fragment from an existing method.

As an example, consider this code:

```
public void report(Writer out, List<Machine> machines) throws IOException {
  for (Machine machine : machines) {
    out.write("Machine " + machine.name());
    if (machine.bin() != null)
      out.write(" bin=" + machine.bin());
    out.write("\n");
  }
}
```

We want to extract the code that reports on each Machine.

Step 1: Create a new, empty method.

```
private void reportMachine() {
}
```

At this stage, we are designing a mini-API. It is important to choose a name that reflects the new method's purpose and not its implementation.
In case of any doubt, we can check that the new method signature is valid by compiling and testing.

In case of any doubt, we can check that the new method signature is valid by compiling and testing.

Step 2: Copy the code into the new method.

This is a simple copy-paste:

```
private void reportMachine() {
  out.write("Machine " + machine.name());
  if (machine.bin() != null)
    out.write(" bin=" + machine.bin());
  out.write("\n");
}
```

In our example, this creates a reportMachine method that doesn't compile, due to the temporary variable machine and the out parameter.

Note that the original method remains unchanged at this point.

Step 3: Add parameters for temporary variables.

For each of the temporary variables used in the copied code we add a parameter to the new method:

```
private void reportMachine(Writer out, Machine machine) {
  out.write("Machine " + machine.name());
  if (machine.bin() != null)
    out.write(" bin=" + machine.bin());
  out.write("\n");
}
```

We also need to declare the checked exception thrown by the write methods:

```
private void reportMachine(Writer out, Machine machine) throws IOException {
  out.write("Machine " + machine.name());
  if (machine.bin() != null)
    out.write(" bin=" + machine.bin());
  out.write("\n");
}
```

At each stage, we can check our progress by compiling. We know we're done when the new method compiles cleanly. And since it still hasn't called, the entire application should pass its tests at this point.

Step 4: Call the new method.

Finally, we can replace the copied code in the original method by a call to the new method:

```
public void report(Writer out, List<Machine> machines) throws IOException {
  for (Machine machine : machines) {
    reportMachine(out, machine);
  }
}
```

Compile, test, and we're done.

The Extract Method refactoring can be a little more complex if the code we want to extract modifies a temporary variable. For example, consider the following modified version of the previous code:

```
public String report(List<Machine> machines) {
  String result = "";
  for (Machine machine : machines) {
    result += "Machine " + machine.name();
    if (machine.bin() != null)
            result += " bin=" + machine.bin();
    result += "\n";
  }
  return result;
}
```

Here, the code we want to extract modifies the result temporary variable. In this case, during Step 3 we need to declare a new result in the new method and return its value at the end of the computation:

```
private String reportMachine(Machine machine) {
  String result = "Machine " + machine.name();
  if (machine.bin() != null)
          result += " bin=" + machine.bin();
  result += "\n";
  return result;
}
```

Now in Step 4 we need to use the returned machine report:

```
public String report(List<Machine> machines) {
  String result = "";
  for (Machine machine : machines) {
          result += reportMachine(machine);
  }
  return result;
}
```

Most of the available refactoring tools provide an automated implementation of this refactoring.

Introduce Parameter Object

A group of parameters is often seen together in method signatures. We can remove some duplication and convert them into a single new domain abstraction.

For this example, consider an application in which a robot moves along a row of machines in a production plant. We may have an object such as this:

```
public class Report {
  public String report(List<Machine> machines, Robot robot) {
    String result = "FACTORY REPORT\n";
    for (Machine machine : machines) {
      result += reportMachine(machine);
    }
    return result + "\n" + reportRobot(robot) + "========\n";
  }
}
```

Together with client code such as this:

```
String report = Report.report(machines, robot);
```

If we notice that the list of Machines is often passed around with the Robot, we may decide to parcel them up together as a Plant object.

Step 1: Create a new class for the clump of values.

First, we create a new Plant class:

```
public class Plant {
  public final List<Machine> machines;
  public final Robot robot;

  public Plant(List<Machine> machines, Robot robot) {
    this.machines = machines;
    this.robot = robot;
  }
}
```

This is a simple container for the two values, and we have made it immutable to keep things clean.

Step 2: Add Plant as an additional method parameter.

We pick any one method that takes machines and robot as parameters, and add an additional parameter for the plant:

```
public class Report {
  public String report(List<Machine> machines, Robot robot, Plant plant) {
    //...
  }
}
```

And change the caller to match:

```
String report = Report.report(machines, robot, new Plant(machines, robot));
```

Compile and test to verify that we have changed no behavior.

Step 3: Switch the method to use the new parameter.

Now we make the original parameters redundant, one at a time. First, we alter the method to get the machines from the plant:

```
public class Report {
  public String report(List<Machine> machines, Robot robot, Plant plant) {
    String result = "FACTORY REPORT\n";
    for (Machine machine : plant.machines) {
      result += reportMachine(machine);
    }
    return result + "\n" + reportRobot(robot) + "========\n";
  }
}
```

This is another safe base; we have the option here to compile, test, and commit if we wish.

Now that the machines' parameter is unused within the method, we can remove it from the signature:

```
public class Report {
  public String report(Robot robot, Plant plant) {
    //...
  }
}
```

And from every call site:

```
String report = Report.report(robot, new Plant(machines, robot));
```

This is another safe base, if we wish to take advantage of it.

Then we do the same with the robot parameter, and we're done:

```
public class Report {
  public String report(Plant plant) {
    String result = "FACTORY REPORT\n";
    for (Machine machine : plant.machines) {
      result += reportMachine(machine);
    }
    return result + "\n" + reportRobot(plant.robot) + "========\n";
  }
}
```

And:

```
String report = Report.report(new Plant(machines, robot));
```

We can follow these same steps for every method that has the same two parameters.

Separate Query from Modifier

Methods that have side effects are harder to test and less likely to be safely reusable. Methods that have side effects and return a value also have multiple responsibilities. So oftentimes it can benefit the code to split such a method into separate query and command methods.

Imagine we have a Meeting class with a method that looks for a manager in its configuration file and sends them an email:

```
class Meeting {
  public StaffMember inviteManager(String fileName) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String line = "";
    while ((line = in.readLine()) != null) {
      StaffMember person = new StaffMember(line);
      if (person.isManager()) {
        sendInvitation(this, person);
        return person;
      }
    }
    return null;
  }
}
```

This method performs as a query – looking up the manager in the file – and as a command. The code will be somewhat more testable if we can separate those two responsibilities.

Step 1: Create a copy with no side effects.

We create a new method by copying the original and deleting the side effects:

```
class Meeting {
  public StaffMember findManager(String fileName) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String line = "";
    while ((line = in.readLine()) != null) {
      StaffMember person = new StaffMember(line);
      if (person.isManager()) {
        return person;
      }
    }
    return null;
  }
}
```

This is a pure query, and as it is never called, we can compile, test, and commit if we wish.

Step 2: Call the new query.

Now that we have the new query method, we can use it in the original method:

```
class Meeting {
  public StaffMember inviteManager(String fileName) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String line = "";
    while ((line = in.readLine()) != null) {
      StaffMember person = new StaffMember(line);
      if (person.isManager()) {
        sendInvitation(this, person);
        return findManager(fileName);
      }
    }
    return null;
  }
```

Compile and test.

Step 3: Alter the callers.

Imagine the original method is called here:

```
public void organiseMeeting() throws IOException {
  StaffMember manager = meeting.inviteManager(employeeData);
  notifyOtherAttendees(manager);
}

}
```

We alter this method to make separate explicit calls to the command and the query:

```
public void organiseMeeting() throws IOException {
  meeting.inviteManager(employeeData);
  StaffMember manager = meeting.findManager(employeeData);
  notifyOtherAttendees(manager);
}
```

We do this for all callers of the original method, and as usual we can compile, test, and commit at any stage because we are not altering the application's overall behavior.

Step 4: Void the command method.

When all the callers have been converted to use the command-query separated methods, we can remove the return value from the original method:

```
class Meeting {
  public void inviteManager(String fileName) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String line = "";
    while ((line = in.readLine()) != null) {
      StaffMember person = new StaffMember(line);
      if (person.isManager()) {
        sendInvitation(this, person);
      }
    }
  }
}
```

This method is now a pure command. As ever, the callers should still pass their tests.

Step 5: Remove duplication.

Finally, in our example we can use the new query within the command method to remove some duplication:

```
class Meeting {
  public void inviteManager(String fileName) throws IOException {
    StaffMember manager = findManager(fileName);
    if (manager != null)
      sendInvitation(this, manager);
  }
}
```

Replace Inheritance with Delegation

Sometimes you need to extract a class from an inheritance hierarchy.

Step 1: Create a new field in the subclass to hold an instance of the superclass.
Initialise the field to this.

Step 2: Change all calls to superclass methods so that they refer to the new field.

Instead of directly calling superclass methods from the subclass, call them via the object referred to in your new field. Compile and test.

Step 3: Remove the inheritance and initialise the field with a new instance of the superclass.

Compile and test. At this point, we may need to add new methods to the subclass if its clients use methods it previously inherited. Add these missing methods, compile, and test.

Remove Control Couple

Sometimes a method parameter is used inside the method solely to determine which of two or more code paths should be followed. Thus the method has at least two responsibilities, and the caller "knows" which one to invoke by setting the parameter to an appropriate value. Boolean parameters are often used in this way.

Step 1: Isolate the conditional
If necessary, use Extract Method to ensure that the conditional check and its branches form the entirety of a method.

Step 2: Extract the branches
Use Extract Method on each branch of the conditional, so that each consists solely of a single call to a new method.

Step 3: Remove the coupled method
Use Inline Method to replace all calls to the conditional method, and then remove the method itself.

Replace Error Code with Exception

Sometimes the special values returned by methods to indicate an error can be too cryptic, and it may be preferable to throw an exception.

Step 1: Decide whether the exception should be checked or unchecked.

Make it unchecked if the caller(s) should already have prevented this condition from occurring.

Step 2: Copy the original method, and change the new copy to throw the exception instead of returning the special code.

This new method is not called yet. Compile and test.

Step 3: Change the original method so that it calls the new one.

The original method will now catch the exception and return the error code from its catch block. Compile and test.

Step 4: Use Inline Method to replace all calls to the original method by calls to the new method.

Compile and test.

Hide Delegate

When one object reaches through another to get at a third, it may be time to reduce coupling and improve encapsulation.

Step 1: For each method you reach through to call, create a delegating method on the middle-man object.

Compile and test after creating each method.

Step 2: Adjust the client(s) to call the new delegating methods.

Compie and test at each change.

Step 3: If possible. remove the accessor from the middle-man.

If no-one now accesses the delegated object via the middle-man, remove the accessor method so that no-one can in the future.

Preserve Whole Object

When several of a method's arguments could be obtained from a single object, it may be beneficial to pass that object instead of the separate values.

Step 1: Add a new parameter for the whole object

Pass in the object that contains the values you wish to replace. At this stage, this extra parameter is not used, so compile and test.

Step 2: Pick one parameter and replace references to it within the method.

Replace uses of the parameter by references to the value obtained from the whole object. Compile and test at each change.

Step 3: Remove the parameter that is now unused.

Don't forget to also delete any code in the callers that obtains the value for that parameter. Compile and test.

Step 4: Repeat for every other value that can be obtained from the new parameter.

Compile and test.

COMMONLY USED REFACTORINGS

A “refactoring” is any behavior-preserving transformation of a software application. That is, a refactoring is any change to your code that doesn't change what it does.

Unfortunately, that definition is so vague as to be nearly useless in practice. In order to become a practical tool, refactoring needs an objective and a set of ready-made shrink-wrapped techniques.

This table shows a small selection of the most well-known refactorings:

Refactoring	Purpose
Extract Method	Turns a code fragment into a method whose name describes its purpose. Helps improve the callers' readability, and may help reduce duplication.
Inline Method	Replaces calls to a method by copies of its body. Brings code into one place in preparation for other refactorings.
Rename Method	Can help client code become more readable. Often used after other refactorings if the method's responsibilities are now better understood.
Introduce Explaining Variable	Save the result of (part of) a complicated expression in a temporary variable. Helps to improve an algorithm's readability.
Inline Temp	Replace uses of a temporary variable by its value. Often used to bring code together in preparation for other refactorings such as Extract Method.
Add Parameter	Passes extra information into a method. Often used during larger code restructuring.
Remove Parameter	Removes a parameter that is no longer used by a method. Helps to keep the method's interface clean and readable for client code.
Extract Class	A class has too many responsibilities, so you split out part of it. The new class's name contributes to the code's domain language; testability and reuse may also be improved.
Introduce Parameter Object	When several methods take the same bunch of parameters, creates a new object that represents the bunch. This is another way of finding new classes and thus new domain concepts.
Introduce Null Object	Replaces null checks by introducing a new class to represent that special case. Helps to remove duplication and simplifies conditional logic.
Replace Magic Number with Symbolic Constant	Gives a name to a literal number. Improves code readability, and can help to identify subtle dependencies between algorithms.

Note that many of these are inverses of each other.

REFACTORING IN PRACTICE

Doing it safely

Refactoring involves making numerous changes to a body of code, and thus it incurs the significant risk of breaking that code. Make sure your code is working before you start. The best way to do that is to have an automated test suite that you trust and gives you good coverage. Use a version control tool and save a checkpoint before each refactoring. Not only does this mean you can recover quickly from disasters, it also means you can try out a refactoring and then back out if you don't like where it took your code. Finally, use a refactoring tool if there is one available for your environment.

Refactoring tools

This is a brief survey of the many refactoring tools available. In most cases they are built into, or are extensions of, IDEs.

Language	Tools
C	Lint Visual Studio
C++	Lint Visual Assist Visual Studio
C#	CodeRush Resharper Visual Studio
Java	Eclipse IntelliJ IDEA Netbeans Oracle JDeveloper
Ruby	Aptana Netbeans Raffle
Smalltalk	Smalltalk Refactoring Browser

CONCLUSION

Refactoring involves making numerous changes to a body of code, and thus it incurs the significant risk of breaking that code.

Hot tips

1. Make sure your code is working before you start.
2. Ensure you have an automated test suite that you trust and gives you good coverage. Run the tests frequently before, during, and after each refactoring.

3. Use a version control tool and save a checkpoint before each refactoring. Not only does this mean you can recover quickly from disasters, it also means you can try out a refactoring and then back it out if you don't like where it took your code.
4. Break each refactoring down into small, safe steps by following the core process outlined above.
5. Finally, use a refactoring tool if there is one available for your environment.

FURTHER READING

Fowler, M et al: Refactoring – Improving the design of existing code, Addison-Wesley 1999.

This is the book that ushered refactoring onto the main stage. Contains strong introductory material and an extensive catalogue of refactorings, with an emphasis on safe, step-by-step recipes that can be followed mechanically. [Code examples in Java.]

Wake, W: Refactoring workbook, Addison-Wesley 2003.
A tour of the major code smells, with hints on how to recognize them and how to remove them, plus scores of exercises and challenges that encourage the reader to practice refactoring on small and large practical examples. [Code samples in Java.]

Kerievsky, J: Refactoring to patterns, Addison-Wesley 2004.
Recipes and worked examples showing how to tackle large, complex refactorings in real-world programs. [Code examples in Java.]
Kernighan, B & Plauger, PJ: Elements of programming style, McGraw-Hill 1974.

Gentle, highly instructive, and of great historical significance, this book pre-dates the notions of refactoring and code smells by several decades, and yet the authors refactor the smells out of several published programs. [Code examples in Fortran, PL/1.]

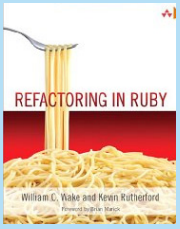
Fields, J et al: Refactoring, ruby edition, Addison-Wesley 2009.
A re-working of Fowler's original for the Ruby language; includes additional Ruby-specific code smells and refactorings. [Code examples in Ruby.]
Meszaros, G: xUnit test patterns: Refactoring test code, Addison-Wesley 2007.

In-depth coverage of an oft-neglected area of refactoring. Includes a catalogue of test smells and sample refactorings. [Code examples in multiple languages.]

ABOUT THE AUTHORS

Kevin Rutherford, PhD, is a UK-based extreme programmer and agile/TDD coach. He is a Chartered Engineer and has over 30 years' industry experience in software development, including spells in project management and as the owner of a successful startup. He is the author of 'Refactoring in Ruby' http://www.refactoringinruby.info and created the Reek codesmell detector. In recent years he has returned to focus on developing great code and great software teams. He can be contacted via http://kevinrutherford.co.uk.

RECOMMENDED BOOK



The book shows you when and how to refactor with both legacy code and during new test-driven development, and walks you through real-world refactoring in detail. The workbook concludes with several applications designed to help practice refactoring in realistic domains, plus a handy code review checklist you'll refer to again and again. Along the way, you'll learn powerful lessons about designing higher quality Ruby software—lessons that will enable you to experience the joy of writing consistently great code.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "'DZone is a developer's dream'," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-66-0
ISBN-10: 1-936502-66-6
50795
9 781936 502660
$7.95

Version 1.0