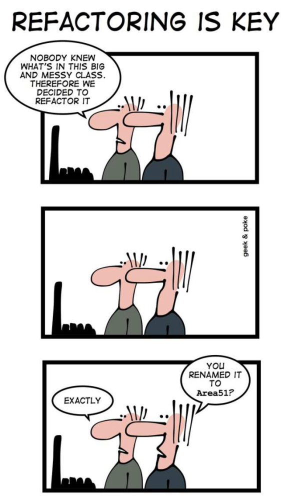

REFACTORING IS KEY
NOBODY KNEW WHAT'S IN THIS BIG AND MESSY CLASS. THEREFORE WE DECIDED TO REFACTOR IT
geek & poke
EXACTLY
YOU RENAMED IT TO Area51?

Common Smells

www.cs.uoi.gr/~zarras/soft-devII.htm

Sources: M. Fowler Refactorings Catalog
W. Wake Refactoring Workbook

Comments

Comments

Symptoms

Scan the text for "//" or "/*" (comment markers).

Causes

Usually, for the best of reasons. The author realizes that **something isn't as clear** as it could be, and adds a comment.

Some comments are particularly helpful:
Legal comments, informative, Javadoc public API, TODO, amplification, warnings. Etc….

Comments

What to Do

When a comment explains a block of code, you can often use ***Extract Method*** *to* pull the block out into a separate method.

The comment will often suggest a name for the new method.

Extract Method

```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

If you have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

Extract Method

Extract Method is one of the most common refactorings

If you have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

Short, well-named methods increase the chances that other methods can reuse them.

It allows the higher-level methods to read more like a story.

Overriding also is easier when the methods are fine grained.

Small methods really work only when you have good names, so you need to pay attention to naming.

Comments

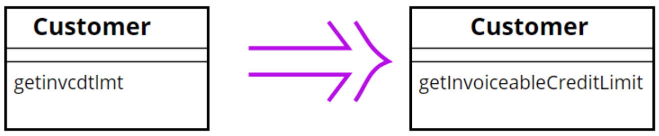
What to Do

When a comment explains what a method does (better than the method's name!), use ***Rename Method*** *using the comment as the basis of the new name*.

Rename Method

The name of a method does not reveal its purpose.

Change the name of the method.



Before changing the name, **check if the method overrides (or is overridden) a super class method (by subclasses methods).** In that case all the names must change.

Changing the name implies changing the callers of the method.

What we do if this is not possible ??

We can **keep the previous version** of the method and **make it call the new one to avoid duplication;** mark the old version of the method as **deprecated**

Comments

What to Do

When a comment explains preconditions, consider using ***Introduce Assertion*** *to* replace the comment with code.

Introduce Assertion

A section of code assumes something about the state of the program.

Make the assumption explicit with an assertion.

```
public void setup() {
         Connection conn = getConnection();
         // conn should be != null to the
         // rest of the code to work…
         …….
}
```

⇓

```
// In Java there is a specific programming construct!!
// Don't confuse it with Junit
// To enable them java -ea

public void setup() {
         Connection conn = getConnection();
         assert conn != null : "Connection is null";
}
```

Introduce Assertion

Often sections of code work only if certain conditions are true. Such assumptions often are not stated but can only be decoded by looking through an algorithm.

Sometimes the assumptions are stated with a comment.

A better technique is to make the assumption explicit by writing an assertion.

Assertions act as communication and debugging aids. In **communication** they **help the reader understand the assumptions** the code is making. In **debugging**, assertions can **help catch bugs** closer to their origin.

Long Method

Long Method

Symptoms

Large number of lines in the method. (immediately suspicious of any method with more than five to ten lines.)

Causes

A method has started down a path, and rather than break the flow or identify the helper objects, the author adds "one more thing."

Code is often easier to write than it is to read, so there's a temptation to write blocks that are too big.

Long Method

What to Do

Use ***Extract Method*** *to break up the method into smaller pieces. Look for comments or* white space that delineate "interesting" blocks.

Long Method

What to Do

You may find **other related refactorings** (those that clean up straight-line code, **conditionals**, and **variable usage**) helpful before you even begin splitting up the method.

Inline Temp

You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.

Replace all references to that temp with the expression.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

⇓

```
return (anOrder.basePrice() > 1000)
```

Replace Temp with Query

You are using a temporary variable to hold the result of an expression.

Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

$\Downarrow$

```
if (basePrice() > 1000)
  return basePrice() * 0.95;
else
  return basePrice() * 0.98;
...
double basePrice() {
  return _quantity * _itemPrice;
}
```

zas14

Replace Temp with Query

Replace Temp with Query often is a vital step before Extract Method.

Local variables *make it* **difficult to extract**, so replace as many variables as you can with queries.

The straightforward cases of this refactoring are those in which temps are assigned only to once. Other cases are trickier (e.g. see Split Temporaty Variable) but possible.

Slide 18

zas14 why is this an issue ??

zarras, 17/05/2012

Split Temporary Variable

You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.

Make a separate temporary variable for each assignment.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

$\Downarrow$

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Split Temporary Variable

Temporary variables are made for various uses.

Some of these uses naturally lead to the temp's being assigned to several times. **Loop variables** change for each run around a loop (such as the i in for (int i=0; i<10; i++). Collecting temporary variables collect together some value that is built up during the method.

Many **other temporaries** are used to hold the result of a bit of code for easy reference later.

These kinds of variables **should be set only once**. That they are set more than once is a **sign that they have more than one responsibility** within the method.

Any variable with more than one responsibility should be replaced with **a temp for each responsibility**. Using a temp for two different things is very confusing for the reader.

Decompose Conditional

You have a complicated conditional (if-then-else) statement.

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
  charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
if (notSummer(date))
  charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Decompose Conditional

One of the most common areas of complexity in a program lies in complex conditional logic. As you write code to test conditions and to do various things depending on various conditions, you quickly end up with a pretty long method. Length of a method is in itself a factor that makes it harder to read, but conditions increase the difficulty.

As with any large block of code, you can make your intention clearer by decomposing it and replacing chunks of code with a method call named after the intention of that block of code.

With conditions you can receive further benefit by doing this for the conditional part and each of the alternatives.

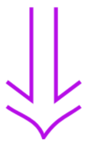
This way you highlight the condition and make it clearly what you are branching on. You also highlight the reason for the branching.

Consolidate Conditional

You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
  if (_seniority < 2) return 0;
  if (_monthsDisabled > 12) return 0;
  if (_isPartTime) return 0;
  // compute the disability amount
```



```
double disabilityAmount() {
  if (isNotEligableForDisability()) return 0;
  // compute the disability amount
```

Consolidate Conditional

Sometimes you see a series of conditional checks in which each check is different yet the resulting action is the same. When you see this, you should use **ands** and **ors** to consolidate them into a single conditional check with a single result.

Consolidating the conditional code is important for two reasons.

First, it makes the check clearer by showing that you are really making a single check. The sequence has the same effect, but it communicates carrying out a sequence of separate checks that just happen to be done together.

The second reason for this refactoring is that it often sets you up for EXTRACT METHOD and DECOMPOSE CONDITIONAL.

The reasons in favor of consolidating conditionals also point to reasons for not doing it. **If you think the checks are really independent and shouldn't be thought of as a single check, don't do the refactoring.**

****** Check that any of the conditionals has side effects. *If there are side effects, you won't be able to do this refactoring***

Consolidate Duplicate Conditional Fragments

The same fragment of code is in all branches of a conditional expression.

Move it outside of the expression.

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

```
if (isSpecialDeal())
  total = price * 0.95;
else
  total = price * 0.98;
send();
```

Consolidate Duplicate Conditional Fragments

Sometimes you find common code executed in all legs of a conditional. In that case you should move the code to outside the conditional. This makes clearer what varies and what stays the same. Simplifies DECOMPOSE CONDITIONAL too.

If the common code is at the **beginning**, move it to **before the conditional**.

If the common code is at the **end**, move it to **after the conditional**.

If the common code is in the **middle**, **look to see whether the code before or after it changes anything**. Depending on this, you can move the common code forward or backward to the ends.

If there is **more than a single statement**, you should **extract that code into a method**.

Replace Nested Conditional with Guard

A method has conditional behavior that does not make clear what the normal path of execution is

Use Guard Clauses for all the special cases

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
    if (_isSeparated) result = separatedAmount();
    else {
      if (_isRetired) result = retiredAmount();
      else result = normalPayAmount();
    };
  }
  return result;
};
```

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

Replace Nested Conditional with Guard

We find that conditional expressions come in two forms.

--- The **first form** is a check whether **either course** is part of the **normal behavior**.
--- The **second form** is a situation in which **one branch** from the conditional indicates **normal behavior** and **the other** indicates an **unusual condition**.

These kinds of conditionals have different intentions, and these intentions should come through in the code.
--- **If both are part of normal behavior**, use a condition with an **if and an else leg**.
--- If the condition is an unusual condition, check the condition and return if the condition is true. This is called **guard clause**.

The key point about *Replace Nested Conditional with Guard Clauses is one of emphasis. If you* are using an if-then-else construct you are giving equal weight to the if leg and the else leg. This communicates to the reader that the legs are equally likely and important. Instead the guard clause says, "This is rare, and if it happens, do something and get out."

Large Class

Large Class

Symptoms

Large number of instance variables
Large number of methods
Large number of lines
Very general name

Causes

A little bit at a time. The author adds "one more capability" to a class, and eventually it grows too big.

Sometimes the problem is a lack of insight into the parts that make up the whole class.

In any case, the class represents too many responsibilities folded into one object.

Large Class

What to Do

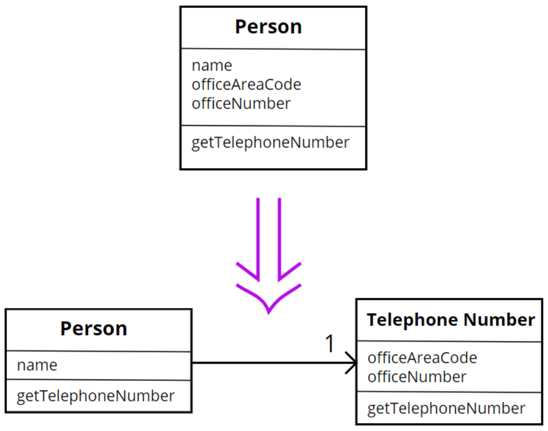
In general, you're trying to break up the class.
***** If the class has Long Methods, address that smell first.**

Extract Class, *if you can identify a new object that has part of this class's* responsibilities

Extract Class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



Extract Class

In practice, classes grow.

You add some operations here, a bit of data there. You add a responsibility to a class feeling that it's not worth a separate class, but as that responsibility grows and breeds, the **class becomes too complicated**.

Such a class is one with many methods and quite a lot of data. A class that is too big to understand easily.

You need to consider where it can be split, and you split it.

A good sign is that **a subset of the data and a subset of the methods seem to go together**. Other good signs are subsets of data that usually **change together** or are particularly dependent on each other.

Large Class

What to Do

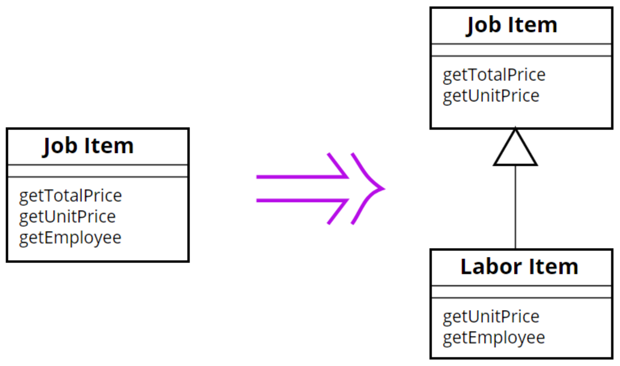
In general, you're trying to break up the class.
If the class has Long Methods, address that smell first.

Extract Subclass*, if you can divide responsibilities between the class and a new* subclass

Extract Subclass

A class has features that are used only in some instances.

Create a subclass for that subset of features.



Large Class

What to Do

Extract Interface,
*you can identify **subsets of features that clients use***

you have no time to extract class or

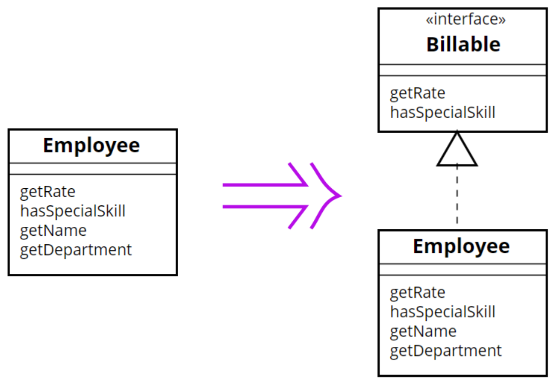
it is very complex to extract class or

It does not make sense to extract class because it is cohesive

Extract Interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.

Extract the subset into an interface.



Extract Interface

z1

Classes use each other in several ways.

Use of a class often means ranging over the whole area of responsibilities of a class.

Another case is use of **only a particular subset of a class's responsibilities by a group of clients**.

For the second case it is often useful to define an interface for the subset of responsibilities.

That way it is easier to **see how the responsibilities of a class divide wrt how they are used by other classes and the different roles played by the class**

Slide 38

z1 this is a good alternative if you dont have time to extract class and reengineer in general
zarras, 30/11/2018

Long Parameter List

Long Parameter List

Symptoms

Count the number of parameters to a method.
(Even three or four might be too many.)

Causes

An author often tries to minimize coupling between objects. Instead of the called object being aware of relationships between objects, you let the caller locate everything; then the method concentrates on what it is being asked to do with the pieces.

Or, the author generalizes the routine to deal with multiple variations: there's a general algorithm and a lot of "control" parameters.

Long Parameter List

What to Do z2

For control parameters ***Replace Parameter with Explicit Methods***

Replace Parameter with Explicit Methods

You have a method that runs different code depending on the values of an enumerated parameter.

Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
  if (name.equals("height")) {
    _height = value;
    return;
  }
  if (name.equals("width")) {
    _width = value;
    return;
  }
  Assert.shouldNeverReachHere();
}
```

$\Downarrow$

```
void setHeight(int arg) {
  _height = arg;
}
void setWidth (int arg) {
  _width = arg;
}
```

Slide 41

z2 when yo ureach the what to do slides go back and forth to explain each bullet

this way ot rolls better!!!

zarras, 30/11/2018

Replace Parameter with Explicit Methods

The usual case for this refactoring is that you have discrete values of a parameter, test for those values in a conditional, and do different things.

Remember, methods should do one thing that can be expressed in a simple TO statement.

Otherwise, the caller has to decide what it wants to do, set the parameter, and generally do more work than needed.

Long Parameter List

What to Do z3

If the parameter value can be obtained from an object or method that the called method already knows, ***Replace Parameter with Method***.

Slide 44

z3 when yo ureach the what to do slides go back and forth to explain each bullet

this way ot rolls better!!!

zarras, 30/11/2018

Replace Parameter with Method

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

Remove the parameter and let the receiver invoke the method.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

$\Downarrow$

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Replace Parameter with Method

If a method can get a value that is passed in as parameter by another means, it should. Long parameter lists are difficult to understand, and we should reduce them as much as possible.

One way of **reducing parameter lists is to look to see whether the receiving method can make the same calculation**.

Long Parameter List

What to Do Z4

If the parameters come from a single object, try ***Preserve Whole Object***.

Preserve Object

You are getting several values from an object and passing these values as parameters in a method call.

Send the whole object instead.

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

$\Downarrow$

```
withinPlan = plan.withinRange(daysTempRange());
```

Slide 47

z4 when yo ureach the what to do slides go back and forth to explain each bullet

this way ot rolls better!!!

zarras, 30/11/2018

Preserve Object

Preserve Whole Object often makes the code more readable.

Long parameter lists can be hard to work with because both caller and callee have to remember which values were there.

There is a **down side**. When you pass in values, the called object has a dependency on the values, but there isn't any dependency to the object from which the values were extracted.

Passing in the required object causes a dependency between the required object and the called object. If this is going to mess up your dependency structure, don't use *Preserve Whole Object.*

Long Parameter List

What to Do z5

If the data is not from one logical object, you still might group them via ***Introduce Parameter Object***.

Slide 50

z5 when yo ureach the what to do slides go back and forth to explain each bullet

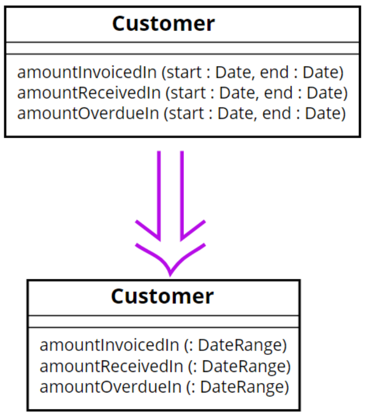
this way ot rolls better!!!

zarras, 30/11/2018

Introduction Parameter Object

You have a group of parameters that naturally go together.

Replace them with an object.


Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)
Customer
amountInvoicedIn (: DateRange)
amountReceivedIn (: DateRange)
amountOverdueIn (: DateRange)

Introduce Parameter Object

Often you see a particular group of parameters that tend to be passed together. Several methods may use this group, either on one class or in several classes.

Such a group of classes is a data clump and can be replaced with an object that carries all of this data. It is worthwhile to turn these parameters into objects just to group the data together.

This refactoring is useful because **it reduces the size of the parameter lists, and long parameter lists are hard to understand**.

You get a deeper benefit, however, because once you have clumped together the parameters, **you soon see behavior that you can also move into the new class**.

Often the bodies of the methods have common manipulations of the parameter values. By moving this behavior into the new object, you can remove a lot of duplicated code.