Volume 3, Issue 10, October 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Making Method Call Simpler in Software Refactoring

Mr. Ganesh B. Regulwar [1] , Dr. R. M. Tugnayat [2]
[1]HOD Department of Computer Science & Engineering
[12] *Shri Shankarprasad Agnihotri College of Engineering, Wardha, MS, INDIA*

Abstract: Refactoring is the art of reworking your code into a more simplified or efficient form in a disciplined way. Refactoring improves internal code structure without altering its external functionality by transforming functions and rethinking algorithms. Consequently, refactoring is an iterative process. By refactoring your JavaScript, Flash, and Java you can streamline its size, simplify your code, and speed up download and execution speed. The majority of information systems these days engender a high level of complexity through the extent of possible inputs to testing, required processing and consequent outputs. In fact, complexity permeates every level of this model for an information system. Complexity thus has a direct effect on the extent to which a system needs to be tested, through those inputs. Complexity also inhibits the ease with which a system can be modified since more time needs to be devoted to assessment of change complexity and resulting tests. Reduction of complexity is the goal of every developer when initially developing a system. So, the code is made simpler by making method calls simpler using several techniques which are described in this paper.

Keywords: improves internal code structure, Reduction of complexity, efforts, software refactoring

1 INTRODUCTION

1.1. Making Method Calls Simpler

Objects are all about interfaces. Coming up with interfaces that are easy to understand and use is a key skill in developing good object-oriented software. This seminar explores refactoring that make interfaces more straightforward. Often the simplest and most important thing we can do is to change the name of a method. Naming is a key tool in communication. If we understand what a program is doing, we should not be afraid to use Rename Method to pass on that knowledge. We can (and should) also rename variables and classes. On the whole these renaming are fairly simple text replacements. Parameters themselves have quite a role to play with interfaces. Add Parameter and Remove Parameter are common refactoring. Programmers new to objects often use long parameter lists, which are typical of other development environments. If we are passing several values from an object, use Preserve Whole Object to reduce all the values to a single object. If this object does not exist, we can create it with Introduce Parameter Object.

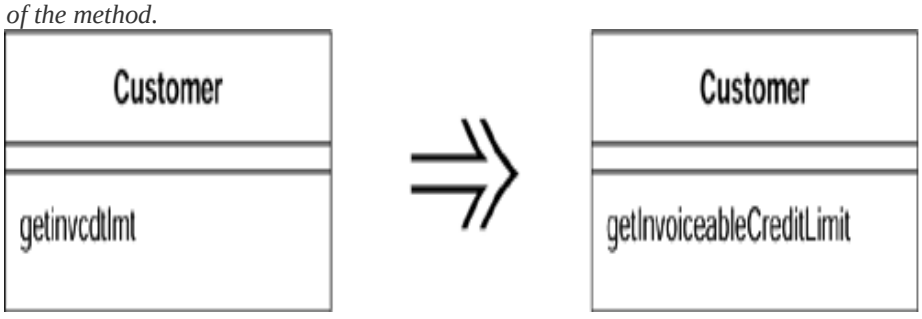
If we can get the data from an object to which the method already has access, we can eliminate parameters with Replace Parameter with Method. If we have parameters that are used to determine conditional behavior, we can use Replace Parameter with Explicit Methods. We can combine several similar methods by adding a parameter with Parameterize Method. One of the most valuable conventions used over the years is to clearly separate methods that change state (modifiers) from those that query state (queries). So whenever they are combined, use Separate Query from Modifier to get rid of them. When refactoring we often need to make things visible for a while and then cover them up with Hide Method and Method. Constructors are a particularly awkward feature of Java and C++, because they force we to know the class of an object you need to create. Often we do not need to know this. The need to know can be removed with Replace Constructor with Factory Method.

Java, like many modern languages, has an exception-handling mechanism to make error handling easier. Programmers who are not used to this often use error codes to signal trouble. We can use Replace Error Code with Exception to use the new exceptional features. But sometimes exceptions aren't the right answer; you should test first with Replace Exception with Test.

1.1.1 Rename Method

The name of a method does not reveal its purpose.

Change the name of the method.



An important part of the code style is small methods to factor complex processes. This can lead you to find out what all the little methods do. The key to avoiding this, is naming the methods. Methods should be named in a way that communicates their intention. A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method. Good naming is a skill that requires practice; improving this skill is the key to being a truly skillful programmer. The same applies to other aspects of the signature. If reordering parameters clarifies matters, does it (see Add Parameter and Remove Parameter).

Example

Here is a method to get a person's telephone number:

```
public String getTelephoneNumber() {
return ("(" + _officeAreaCode + ") " + _officeNumber);}
```

We want to rename the method to getOfficeTelephoneNumber. By creating the new method and copying the body over to the new method, the old method now changes to call the new one:

```
class Person...
public String getTelephoneNumber(){
 return getOfficeTelephoneNumber();
}
public String getOfficeTelephoneNumber() {
return ("(" + _officeAreaCode + ") " + _officeNumber);}
```

- Now we find the callers of the old method, and switch them to call the new one. When we have switched them all, we can remove the old method. The procedure is the same if we need to add or remove a parameter.
- If there aren't many callers, we can change the callers to call the new method without using the old method as a delegating method. If the tests throw a wobbly, then back out and make the changes the slow way.

1.1.2 Add Parameter

- A method needs more information from its caller. So, we need to *add a parameter for an object that can pass on this information.*

- *Add Parameter* is a very common refactoring, one that you almost certainly have already done. You have to change a method, and the change requires information that wasn't passed in before, so you add a parameter.
- Often you have other alternatives to adding a parameter. If available, these alternatives are better because they don't lead to increasing the length of parameter lists. Long parameter lists smell bad because they are hard to remember and often involve data clumps.

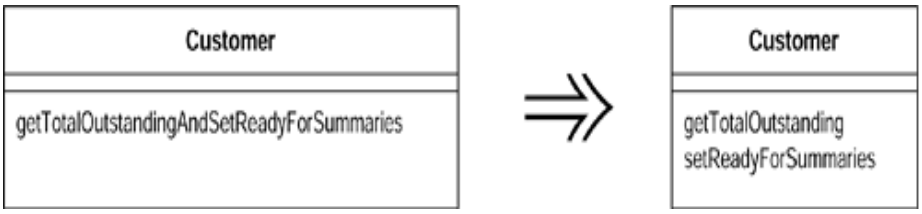
1.1.3 Remove Parameter

- A parameter is no longer used by the method body. It needs to be *removed.*

- Programmers often add parameters but are reluctant to remove them. After all, a spurious parameter doesn't cause any problems, and you might need it again later.
- A parameter indicates information that is needed; different values make a difference. Your caller has to worry about what values to pass. By not removing the parameter you are making further work for everyone who uses the method. That's not a good trade-off, especially because removing parameters is an easy refactoring.
- The case to be wary of here is a polymorphic method. In this case you may well find that other implementations of the method do use the parameter. In this case you shouldn't remove the parameter. You might choose to add a separate method that can be used in those cases, but you need to examine how your callers use the method to see whether it is worth doing that. If some callers already know they are dealing with a certain subclass and doing extra work to find the parameter or are using knowledge of the class hierarchy to know they can get away with a null, add an extra method without the parameter. If they do not need to know about which class has which method, the callers should be left in blissful ignorance.

1.1.4 Separate Query from Modifier

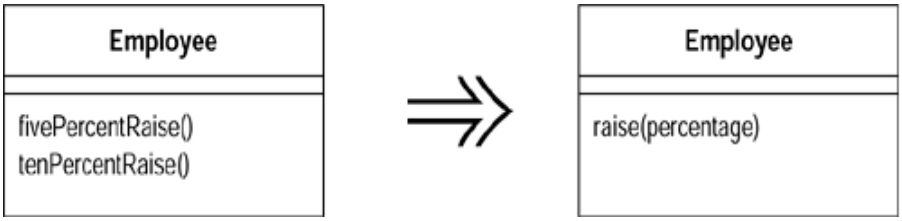
- We have a method that returns a value but also changes the state of an object. For this *create two methods, one for the query and one for the modification.*



- When you have a function that gives us a value and has no observable side effects, we have a very valuable thing. can call this function as often as you like. We can move the call to other places in the method.
- It is a good to clearly signal the difference between methods with side effects and those without. A good rule to follow is to say that any method that returns a value should not have observable side effects. Some programmers treat this as an absolute rule [Meyer].
- If you come across a method that returns a value but also has side effects, you should try to separate the query from the modifier.
- A common optimization is to cache the value of a query in a field so that repeated calls go quicker. Although this changes the state of the object with the cache, the change is not observable. Any sequence of queries will always return the same results for each query [Meyer].

1.1.5 Parameterize Method

- Several methods do similar things but with different values contained in the method body. So, for this *create one method that uses a parameter for the different values.*



- We see a couple of methods that do similar things but vary depending on a few values. In this case you can simplify matters by replacing the separate methods with a single method that handles the variations by parameters. Such a change removes duplicate code and increases flexibility, because you can deal with other variations by adding parameters.

Example

- The simplest case is methods along the following lines:

```
class Employee {
void tenPercentRaise () {
salary *= 1.1;
}
void fivePercentRaise () {
salary *= 1.05;
}
this can be replaced with
void raise (double factor) {
salary *= (1 + factor);}
```

1.1.6 Replace Parameter with Explicit Methods

- We have a method that runs different code depending on the values of an enumerated parameter.So, *create a separate method for each value of the parameter.*
- *Replace Parameter with Explicit Methods* is the reverse of Parameterize Method. The usual case for the former is that you have discrete values of a parameter, test for those values in a conditional, and do different things. The caller has to decide what it wants to do by setting the parameter, so you might as well provide different methods and avoid the conditional. You not only avoid the conditional behavior but also gain compile time checking. Furthermore your interface also is clearer. With the parameter, any programmer using the method needs not only to look at the methods on the class but also to determine a valid parameter value. The latter is often poorly documented.
- The clarity of the explicit interface can be worthwhile even when the compile time checking isn't an advantage. Switch.beOn() is a lot clearer than Switch.setState(true), even when all are doing is setting an internal boolean field.
- We shouldn't use *Replace Parameter with Explicit Methods* when the parameter values are likely to change a lot. If this happens and we are just setting a field to the passed in parameter, use a simple setter.

Example

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

⇓

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

1.1.7 Preserve Whole Object

- We are getting several values from an object and passing these values as parameters in a method call. So *send the whole object instead.*
- This type of situation arises when an object passes several data values from a single object as parameters in a method call. The problem with this is that if the called object needs new data values later, you have to find and change all the calls to this method. You can avoid this by passing in the whole object from which the data came. The called object then can ask for whatever it wants from the whole object.
- In addition to making the parameter list more robust to changes, *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. They also encourage duplicate code because the called object can't take advantage of any other methods on the whole object to calculate intermediate values.
- 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.*
- Another reason I have heard for not using *Preserve Whole Object* is that when a calling object need only one value from the required object, it is better to pass in the value than to pass in the whole object. One value and one object amount to the same thing when you pass them in, at least for clarity's sake (there may be a performance cost with pass by value parameters).
- A called method uses lots of values from another object is a signal that the called method should really be defined on the object from which the values come. When you are considering *Preserve Whole Object,* consider Move Method as an alternative.
- We may not already have the whole object defined. In this case we need Introduce Parameter Object.
- A common case is that a calling object passes several of its *own* data values as parameters. In this case you can make the call and pass in this instead of these values, if you have the appropriate getting methods and you don't mind the dependency.

Example

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

⇓

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

1.1.8 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.Then *remove the parameter and let the receiver invoke the 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. If an object is calling a method on itself, and the calculation for the parameter does not reference any of the parameters of the calling method, you should be able to remove the parameter by turning the calculation into its own method. Also if you are calling a method on a different object that has a reference to the calling object.
- We can't remove the parameter if the calculation relies on a parameter of the calling method, because that parameter may change with each call (unless, of course, that parameter can be replaced with a method). You also can't remove the parameter if the receiver does not have a reference to the sender, and you don't want to give it one.
- In some cases the parameter may be there for a future parameterization of the method. In this case I would still get rid of it. Deal with the parameterization when you need it; you may find out that you don't have the right parameter anyway. I would make an exception to this rule only when the resulting change in the interface would have painful

consequences around the whole program, such as a long build or changing of a lot of embedded code. If this worries you, look into how painful such a change would really be. You should also look to see whether you can reduce the dependencies that cause the change to be so painful. Stable interfaces are good, but freezing a poor interface is a problem.

Example

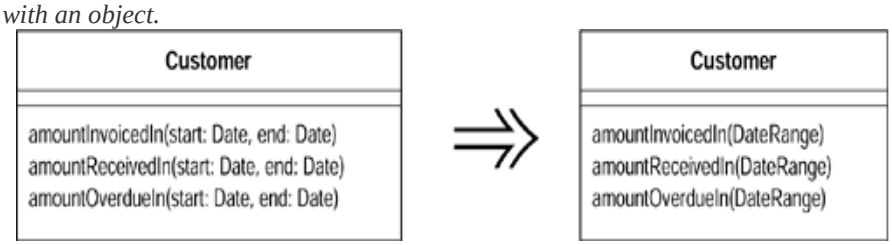
```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

⇓

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

1.2. Introduce Parameter Object

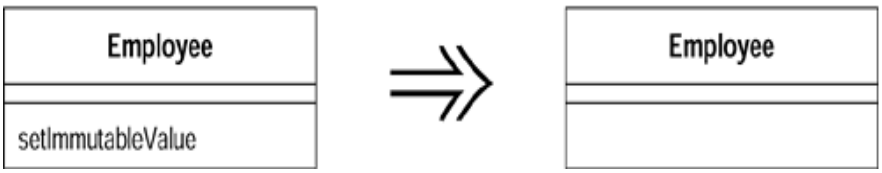
- You have a group of parameters that naturally go together.
- *Replace them with an object.*



- 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. The defined accessors on the new object also make the code more consistent, which again makes it easier to understand and modify.
- 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.

1.2.1 Remove Setting Method

- A field should be set at creation time and never altered.
- *Remove any setting method for that field.*



- Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created, then don't provide a setting method (and make the field final). That way your intention is clear and you often remove the very possibility that the field will change.
- This situation often occurs when programmers blindly use indirect variable access [Beck]. Such programmers then use setters even in a constructor. I guess there is an argument for consistency but not compared with the confusion that the setting method will cause later on.

1.2.2 Hide Method

- A method is not used by any other class.
- *Make the method private.*

- Refactoring often causes you to change decisions about the visibility of methods. It is easy to spot cases in which you need to make a method more visible: another class needs it and you thus relax the visibility. It is somewhat more difficult to tell when a method is too visible. Ideally a tool should check all methods to see whether they can be hidden. If it doesn't, you should make this check at regular intervals.

- A particularly common case is hiding getting and setting methods as you work up a richer interface that provides more behavior. This case is most common when you are starting with a class that is little more than an encapsulated data holder. As more behavior is built into the class, you may find that many of the getting and setting methods are no longer needed publicly, in which case they can be hidden. If you make a getting or setting method private and you are using direct variable access, you can remove the method.

1.2.3 Replace Constructor with Factory Method

- We want to do more than simple construction when we create an object.
- *Replace the constructor with a factory method.*
- The most obvious motivation for *Replace Constructor with Factory Method* comes with replacing a type code with subclassing. We have an object that often is created with a type code but now needs subclasses. The exact subclass is based on the type code. However, constructors can only return an instance of the object that is asked for. So we need to replace the constructor with a factory method.
- We can use factory methods for other situations in which constructors are too limited. Factory methods are essential for Change Value to Reference. They also can be used to signal different creation behavior that goes beyond the number and types of parameters.

Example

```
Employee (int type) {
_type = type ;}
```

```
static Employee create(int type) {
return new Employee(type);}
```

1.2.4 Encapsulate Downcast

- A method returns an object that needs to be downcasted by its callers.
- *Move the downcast to within the method.*

```
Object lastReading() {
return readings.lastElement();}
```

```
Reading lastReading() {
return (Reading) readings.lastElement();}
```

- Downcasting is one of the most annoying things you have to do with strongly typed OO languages. It is annoying because it feels unnecessary; we are telling the compiler something it ought to be able to figure out itself. But because the figuring out often is rather complicated, we often have to do it ourself. This is particularly prevalent in Java, in which the lack of templates means that we have to downcast whenever we take an object out of a collection.
- Downcasting may be a necessary evil, but we should do it as little as possible. If we return a value from a method, and we know the type of what is returned is more specialized than what the method signature says, we are putting unnecessary work on our clients. Rather than forcing them to do the downcasting, we should always provide them with the most specific type we can.
- Often we find this situation with methods that return an iterator or collection.

Example

There a method called lastReading, which returns the last reading from a vector of readings:

```
Object lastReading() {
return readings.lastElement();
}
```

We should replace this with

```
Reading lastReading() {
return (Reading) readings.lastElement();
}
```

- A good lead-in to doing this is where I have collection classes. Say this collection of readings is on a Site class and I see code like this:

```
Reading lastReading = (Reading) theSite.readings().lastElement()
```

I can avoid the downcast and hide which collection is being used with

```
Reading lastReading = theSite.lastReading();
class Site...
Reading lastReading() {
return (Reading) readings().lastElement();}
```

- Altering a method to return a subclass alters the signature of the method but does not break existing code because the compiler knows it can substitute a subclass for the superclass. Of course you should ensure that the subclass does not do anything that breaks the contract of the superclass

1.2.5 Replace Error Code with Exception

- A method returns a special code to indicate an error.

- *Throw an exception instead.*
- In computers, as in life, things go wrong occasionally. When things go wrong, we need to do something about it. In the simplest case, we can stop the program with an error code. This is the software equivalent of committing suicide. If the cost of a program crash is small and the user is tolerant, stopping the program is fine. However, more important programs need more important measures.
- The problem is that the part of a program that spots an error isn't always the part that can figure out what to do about it. When such a routine finds an error, it needs to let its caller know, and the caller may pass the error up the chain. In many languages a special output is used to indicate error. Unix and C-based systems traditionally use a return code to signal success or failure of a routine.
- Java has a better way: exceptions. Exceptions are better because they clearly separate normal processing from error processing. This makes programs easier to understand.

Example

```
int withdraw(int amount) {
  if (amount > _balance)
    return -1;
    else {
  _balance -= amount;
    return 0;
    }}
```

⇓

```
void withdraw(int amount) throws BalanceException {
if (amount > _balance) throw new BalanceException();
  _balance -= amount ;}
```

1.2.6. Replace Exception with Test

- We have to throw a checked exception on a condition the caller could have checked first.
- *Change the caller to make the test first.*
- Exceptions are an important advance in programming languages. They allow us to avoid complex codes by use of Replace Error Code with Exception. Like so many pleasures, exceptions can be used to excess, and they cease to be pleasurable. Exceptions should be used for exceptional behavior—behavior that is an unexpected error. They should not act as a substitute for conditional tests. If we can reasonably expect the caller to check the condition before calling the operation, we should provide a test, and the caller should use it.

Example

```
double getValueForPeriod (int periodNumber) {
    try {
  return _values[periodNumber];
} catch (ArrayIndexOutOfBoundsException e) {
    return 0;
    }}
```

⇓

```
double getValueForPeriod (int periodNumber) {
  if (periodNumber >= _values.length) return 0;
    return _values[periodNumber];}
```

2. ADVANTAGES

- One of the important things about objects is that they allow you to change the implementation of a software module separately from changing the interface. You can safely change the internals of an object without anyone else's worrying about it, but the interface is important—change that and anything can happen.
- Eliminates the need for rigorous (often Error prone) up - front design

3. DISADVANTAGES

- Something that is disturbing about refactoring is that many of the refactorings do change an interface. Something as simple as Rename Method is all about changing an interface.
- There is no problem changing a method name if you have access to all the code that calls that method. Even if the method is public, as long as you can reach and change all the callers, you can rename the method. There is a problem only if the interface is being used by code that you cannot find and change. When this happens, you need a somewhat more complicated process.

4. CONCLUSION

Refactoring clarifies your code into a more efficient form. By transforming your code with refactoring techniques it will be faster to change, execute, and download. Refactoring is an excellent best practice to adopt for programmers wanting to improve their productivity**.** Making method calls simpler we can make our code easy, not necessary the code is reduced by length but it will make it easier to understand.

REFERENCES

[1] Opdyke WF (1992) Refactoring object-oriented frameworks. University of Illinois. Urbana-Champaign, USA.
[2] Griswold WG (1991) Program restructuring as an aid to software maintenance. University of Washington, USA.
[3] Roberts D (1999) Practical Analysis for Refactoring. University of Illinois. Urbana-Champaign, USA.
[4] Tichelaar S (2001) Modeling object-oriented software for reverse engineering and refactoring. University of Bern, Switzerland.
[5] Griswold WG, Notkin D (1983) Automated assistance for program restructuring. ACM Trans. Software Engineering and Methodology, 2(3):228-69.
[6] Bergstein PL (1997) Maintenance of object-oriented systems during structural evolution-TAPOS J. 3(3):185-212.
[7] Roberts D, Brant J, Johnson RE (1997) A refactoring tool for Smalltalk. TAPOS J. 3(4): 253-63.
[8] Mens T, Tourwé T (2004) A survey on software refactoring. IEEE Transactions on Software Engineering 3(2): 126-33.

