The Benefits And Reasons for Doing Refactoring

CSC508 Software Engineering
by Magnus Aase
December 11, 2001

The people behind the work of Refactoring seem all to agree on that Refactoring is not a cure for all software ills. They say "It is no silver bullet" referring to the famous essay by Frederick Brooks, "No Silver Bullet – Essence and Accident in Software Engineering", *The Mythical Man-Month, 1986.* In this essay Brooks is referring to the Silver Bullet as the only bullet that can kill a werewolf. Brooks is saying that there is no such thing as a Silver Bullet in software processes, there is no *one* special solution to solve all our software problems. One size does not fit all. Unfortunately this goes for Refactoring as well. However, I will here look at why Refactoring is still a valuable activity that should be used for several purposes in software development. I will look at following reasons and analyze them more in depth in a later section:

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You to Find Bugs
- Refactoring Increases The Quality Of The Software
- Refactoring As an Educational Tool
- Refactoring Increases Morale
- Refactoring Helps You to Program Faster (…in the end)

All of these above reasons are overlapping each other in one or several ways. However, I will come back to that discussion later. This chapter is just describing what is claimed about Refactoring.

Refactoring Improves the Design of Software

People tend to do changes to realize short-term goals [8]. The trade-off: redesigning causes short-term pain for a longer term gain, is neglected. Developers avoid short-term pains. It now becomes harder to see and understand the design by reading the code. Refactoring is trying to help with this, by shapening up the code. Work is done to remove bits that are not really in the right place. Regular Refactoring helps code retain its shape.

So how does Refactoring improve the design? One important way is to eliminate duplicate code to ensure the code says everything once and only once, which is an essence of good design. Poorly designed code often takes more code to do the same things, because the code quite literally does the same thing in several places. We want to avoid redundancy in our code. The importance of this lies in future modifications to the code. The more code there is, the harder it is to modify correctly. There is simply more code to understand. Example, you change this block of code here, but the system does not do what you expected, because you had to change the code in three other places as well.

Refactoring and design will also be discussed in a later section. But for now, Refactoring helps to take the depreciated code back into a useful and good design, by applying the necessary refactorings step after step with a lot of testing in between.

Refactoring Makes Software Easier to Understand

Programming is in many ways a conversation with a computer. You write code that tells the computer what to do, and it responds by doing exactly what you tell it to do. Hopefully, in time you close the gap between what you want it to do and what you tell it to do. Programming in this mode is all about saying exactly what you want. But then there is another user of your source code. Someone will try to read your code in a few months time to make some changes. So if that programmer takes a week to do that change instead of one hour, that matters. The trouble from my own experience, however, is that when you are trying to get the program to what you want, you are not thinking about that future developer. You are just too focused to get that darn thing to work. Refactoring helps you to make changes that make the code easier to understand. When Refactoring you have the code that works but is not ideally structured. Some time spent Refactoring can make the code better communicate its purpose. Programming in this mode is all about saying what you want.

Refactoring can also help on program understanding for new hires.[15]. When introducing a project to new hires or people new to the project, doing refactoring can help them to understand how the system works. To improve the design of the existing working system, they will have to figure out what to improve, and why the new changes will improve the system.

When refactoring one changes the code to reflect a better understanding, some of the ways to do this are explained in the *How to Refactor?* chapter. One example would be to replace comments and the code with a new self-explaining method. To have the code communicate its purpose and intention without the use of commenting makes it faster and easier to read. When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous. [1]

Refactoring Helps You to Find Bugs

Help in understanding the code also helps in spotting bugs. When one work with and refactor code, one gets a deeper understanding of what the code does, and the understanding is put right back in the code. Along with XP's pair programming refactoring is a good tool for finding bugs in a systematic way.

Refactoring Increases The Quality Of The Software

Software Quality has many connotations that I will not go deeper into right now. However, in this context, increasing the quality of the software contains methods for making maintenance and adding functionality easier. By applying refactoring methods on software, the goal is to improve the design of existing code. Hence, the improvement make the basis for better quality. I will try to discuss this further later.

Refactoring As an Educational Tool

Having become familiar and experienced with use of refactorings and their qualities, this skill has now become a part of your development skills. The more you do refactoring, the more you learn about bad designs that you had to refactor and how you would have done it instead.

Refactoring Increases Morale [15]

According to Dan Stearns , professor at Computer Science, California Polytechnic State University, Refactoring increases morale. His experience was from reverse engineering, which had many similarities to refactoring. I will try to explain this in another chapter. Refactoring increased the morale because the system was to be improved. By having the system improved in such manner, the team knew it was beneficial. The result was a system that was easier to understand and to work on.

Refactoring Helps You to Program Faster

In the end, all earlier points come down to this: Refactoring helps you develop code more quickly. This might sound counterintuitive, because refactoring does not add functionality to the system. When Fowler talks about refactoring, he says people can see that refactoring improves the quality. Improving design, improving readability, reducing bugs, all improves the quality. What quality of code is, will be discussed later.

Insufficiencies with Refactoring and When to Avoid Using It.

Now that I have looked at the benefits and when to refactor, it could also be useful to take a look at the other side. I will here take a look at the problems and some addressed issues with Refactoring. The more discussion about it and its trade-offs will be discussed later.

“When you learn a new technique that greatly improves your productivity, it might be hard so see when it does not apply. Usually you learn it within a specific context, often just a single project. It might be hard to see what causes the technique to be less effective, even harmful.” - *Martin Fowler*

Problems with refactoring and to know its limitations, is the same as with objects in the early ninetiees according to Fowler. It was not that one didn’t think objects had limitations, it was just that one didn’t know what the limitations were. This can be compared to Refactoring. Since Refactoring is such a new concept, there has not been enough experience about it to see where the limitations apply. Therefore, as more people learn about refactoring, we will learn more by monitoring its progress.

In the essay about a 1960’s experience “Plan to throw one away” Fred Brooks arguments: “Plan to throw one away; you will anyway”. In most projects, the first system is barely usable. He says it might be too slow, too big, awkward to use, or all three of these. There is no alternative but to start again, this time smarter and build a redesigned version in which these problems are solved. He arguments that from all large-system experience shows that it wil be done and it is really hard get it right the first time.

Refactoring can suffer from this too. Refactoring can get a great deal out of decayed software but it has no magically powers [1] If the concerned code is neither able to compile or to run in a stable manner, it might be better to throw it away and rewrite the software from scratch. Then next time one will use refactoring to avoid the mistakes made earlier.

When doing refactoring, one should stop when arriving close to a deadline. At that point the productivity gain from refactoring would appear after the deadline and thus to be too late. Ward Cunningham [14] describes unfinished refactoring as going into debt. Most companies need some debt in order to function efficiently. However, with debt come interest payments, that is, the extra cost of maintenance and extension caused by overly complex code. You can bear some interest payments, but if the payments become too great, you will be overwhelmed. It is therefore important to manage your debt, paying parts of it off by means of refactoring. Then getting close to the deadline might be less painful.

Dealing with the insufficiencies of refactoring, is for most of us a learning experience as mentioned by Fowler earlier. Therefore, unfortunately, only a couple of issues have been adressed so far.

Databases.
Many business applications are tightly coupled to the database schema that supports them. When working with relational databases refactoring that move the data to another place are very costly to implement as the database schemas have to be modified as well. Therfore refactoring is hard to do. Data migration is another reason. Even by carefully layering out the system to minimize the dependencies between the database schema and the object model, changing the database schema forces you to migrate the data, which can be a long and painful task when it has to be done manually.

Changing Interfaces
In the OO world we can change the implementation of a software module seperately from changing the interface. Changes to the internals of an object can safely be done without anyone else worrying about it. However, by changing the interface, in the case where the users have only access to the interface (and not the implemented object), everybody needs to change their calls to it.

Collective ownership [Beck] tries to solve this by having collective ownership of the interfaces. Then you will have a copy of the interface for your own use and collaborate on the changes with the other users when done. So, one should try to avoid publishing interfaces prematurely, and rather modifying the code ownership.

Optimized code
Optimized code is another topic which can make refactoring difficult. Soloway [17], note that reviewers have difficulty reviewing and understanding code that has been optimized. To assist in code reviews and walktroughs, the unoptimized code sections might be shown in the refinement of the desgin representation aling with mapings to the actual code. Hence, for walkthroughs, code reviews, optimized code is not preferred due to its readability. It is simply hard to read and understand optimized code. This goes for the process of debugging optimized code as well [24].

When to Refactor?

Opposed to setting aside time every other week or the end of the day, refactoring is an activity that you do all the time in little bursts [Fowler]. You don't decide to refactor, but because you want to do something else.

There are three general cases for when to do Refactoring.

- **Refactor Before You Add Functionality**
    - Improve the design for a better understanding.
    - Then add functionality.

When introduced to new code, the first reason to refactor is to help understanding. If the code is perplexing, then one should do refactoring to make it clearer. Improve the design. Once the design is refactored, adding features can go more quickly and smoothly.

- **Refactor When You Need to Fix a Bug**
    - Improve the design for a better understanding.
    - Then remove bug.

If you get a bug report, then it could be a sign for refactoring, because the code was not clear enough for you to see there was a bug.

- **Refactor As You Do Code Reviews**
    - Improve the design for a better common understanding.
    - Make suggestions for changes
    - Agree on a solution.

Having knowledge about Refactoring, knowing what to change to improve the code, can be a good tool for code reviews. The suggested way by Fowler, is to one reviewer and the original author work on the code together. The reviewer suggests changes, and they both decide whether the changes can be refactored in. If so, they make the changes. This is also one of the main ideas from XP and pair-programming. [2]

Now that we have looked at the idea of in which working situation you would be doing refactoring, the more challenging task for the developer is to figure out where in the code that should be refactored. So, when working with the code, when do you know you have to change it?

Answer: This is a process often based on human intuition and experience.[1]
"*Bad smells in code*" is a notion for the process when to refactor explained by Kent Beck [2]. This process looks for certain familiar structures and patterns in the code that suggests the possibility for refactoring. To figure out when to refactor, one has to first localize and identify the problem. This problem characterized as the "bad smell". The following bad smells are describing a part of a larger catalogue [1][2] based on comprehensive programming experience in the industry.

Since the list of bad smells is rather long, I will here just look at some of the of them. The italic method names, are names of the refactorings approaching the smells explained in the chapter "*How to Refactor"*.

Duplicate Code

By saying everything once and only once, comes the problems of duplication. The number one in the stink parade. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to combine them.

Example. The simplest duplicated code problem is when one have the same expression in two methods of the same class. Then all one have to do is the *Extract Method* and join the code from both places.

Another common duplication problem is when one have the same expression in two sibling subclasses. One can eliminate this duplication by using the *Extract Method* in both classes then *Pull Up Field*. If the code is similar but not the same, you need to use *Extract Method* to separate the similar bits from the different ones. If then the methods do the same thing with a different algorithm, you can choose the clearer of the two algorithms and use *Substitute Algorithm*.

Long Method

The object programs that live best and longest are those with short methods [1]. The longer a procedure is, the more difficult it is to understand. Another way for easier understanding is good naming. This method is aggressive about decomposing methods, and the heuristic is whenever one feel to comment something, write a method instead. Such a method contains the code that was commented but it is named after the intention of the code rather than how it is done. The key here is to narrow the semantic distance between what the method does and how it does it.

Most of the times, this method is using the *Extract Method*, find parts of the method that seem to go nicely together and make a new method. However, if you have a lots of parameters and temporary variables, these elements get in the way of extracting methods. What happens is that using *Extract Method* ends up passing the parameters and variables to the extracted method making it even less readable than the original. To overcome this problem, one can use the *Replace Temp with Query* to eliminate the temporary variables and the long lists of parameters can be slimmed down by using *Introduce Parameter Object* and *Preserve Whole Object.*

Large class

When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code are often occurring.

One can *Extract Class* to bundle a number of the variables. Choose variables to go together in the component that makes sense for each. For example, "depositAmount" and "depositCurrency" are to variables likely to belong together in a component. Also, if the component make sense as a subclass, using *Extract Subclass* could be useful.

Fowler claims that with a class with too many instance variables, and a class with too much code is prime breeding ground for duplicated code, chaos and death. A simple solution is to eliminate redundancy in the class itself, If one have five hundred-line methods with lots of code in common, one may turn them into 5 ten-line methods with another 10 two-line methods extracted from the original.

A more specific example would be a class that contain a user interface (the GUI), so the class is representing the model and the display component.

So, for a class with a huge pile of variables, the usual solution for a class with too much code is either to *Extract Class* or *Extract Subclass*.

Long Parameter List
In the earlier programming days one were taught to pass in all needed parameters by a routine [Fowler]. This was understandable because the alternative was global data, and global data is evil and usually painful. Objects change this situation because if one don't have something one needs, one can always ask another object to get it for you. So with objects one doesn't pass in everything the method needs; instead one pass enough so that the method can get to everything it needs. Objects are passed instead. A lot of what a method needs is available in the method's host class.

To reduce long parameter lists, however, one can use *Replace Parameter with Method* when you can get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter. Use *Preserve Whole Object* take a bunch of data from an object and replace it with the object itself.

Divergent Change
Divergent Change occurs when one class is commonly changed in different ways for different reasons. E.g "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time I get a new financial instrument". This is a situation where two objects are better than one. Solution to this is to identify everything that changes for a particular cause and use *Extract Class* to put them all together.

Shotgun Surgery
This smell is the similar to Divergent Change but the is of opposite character. Divergent Change is one class that suffers many kinds of changes, and Shotgun Surgery is one change that alters many classes.

So for this case, one wants to use the *Move Method* and *Move Field* to put all the changes into a single class. If no current class look like a good candidate, create one.

Feature Envy
This smell identifies a method that seems more interesting in a class other than its own. A common focus the envy is the data. The method here invokes many getting methods on another object to calculate some value. The method wants therefore wants to be elsewhere, so the cure for this one is to use the *Move Method* to get there. It is not always clear when to do this, but the heuristic is to determine which class has most data and put the method with that data.

Data Clumps
Here we have the smell of identifying several data items, Data Clumps that seem to hang together in lots of places: fields in a couple of classes, parameters in many method signatures. Bunches of data that hang around together should be made into their own object.

The first step is to look for where the clumps appear as fields. Use *Extract Class* on the fields to turn the clumps into an object. Then turn the attention to method signatures using *Introduce Parameter Object* or *Preserve Whole Object* to slim them down. The benefit is that one can shrink a lot of parameter lists and simplify method calling. One can now look for cases of Feature Envy, which will suggest behavior that can be moved into you new classes.

Switch Statements
The problem with switch statements is mostly that of duplication. Often one finds the same switch statement scattered about in a program in different places. If one add a new clause to the switch, you have to find all these switch, statements and change them. The object-oriented notion of polymorphism (to have multiple forms), gives a way to deal with this problem.

Most times when seeing a switch statement, one should consider polymorphism. The issue is where the polymorphism should occur. Often the switch statement switches on a type code. One wants the method or class that hosts the type code value. Use *Extract Method* to extract the switch statement and then *Move Method* to get into the class where the polymorphism is needed. But if one only has a few cases that affect a single method, and one doesn't expect them to change, then polymorphism is overkill.

Lazy Class
Smell for classes that isn't doing enough to pay for itself. These classes should be eliminated. If they are nearly useless, use the *Inline Class*.

Speculative Generality
This smell concerns whereabouts generalities. This is code might be useful in the future that includes all sorts of hooks and special cases to handle things that aren't required. The result of this is often harder to understand and maintain. If it is not being used and it is not in the way, get rid of it.

Message Chains
One sees message chains when a client asks one object from another object, which the client then asks for yet another object, which the client then asks for yet another object, and so on. This is a long line of getThis methods. Navigating this way may means the client is coupled and dependent to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.

Middle Man
Smell for the Middle Man. Example. A class's interface has half of the methods are delegating to another class. The extra communication here is unnecessary, so use *Remove Middle Man* and talk to the object that really knows what is going on.

Data Class
These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. Look for where these getting and setting methods are used by other classes. Try to use *Move Method* to move behavior into the data class for more meaning.

Refused Bequest
Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? They are given all these great gifts and pick just a few to play with.

The traditional thinking is that the hierarchy is wrong. One needs to create a new sibling class and use *Push Down Method* and *Push Down Field* to push all the unused methods to the sibling. That way the parent holds only what is common. Often you'll hear advice that all superclasses should be abstract.

Comments
The smell here is for figuring out when comments are superfluous and are describing more *what* and *how* of the code rather than *why.* If the comments are explaining *what* the block of code is doing, try *Extract Method* so the comments are no longer needed.

A good time to use a comment, is when one doesn't know what to do. In addition to describing what is going on, comment can indicate areas in which one aren't sure. This kind of information helps future modifiers, especially forgetful ones.

How to refactor?

There are currently two ways to do Refactoring. The first is refactoring manually, the second relies on tools. I will here try to see how they are to be done.

Refactoring Manually

In these following pages I have chosen to look at 15 refactorings from Martin Fowlers catalogue of 75 refactorings. They all tie together with the explanations of the "Bad Smells" under the "*When to Refactor*" chapter.

Martin Fowler [1] is aware of the fact that his catalog is by now no complete collection of sensible refactorings:

> As you use the refactorings bear in mind that they are a starting point. You will doubtless find gaps in them. I'm publishing them now because although they are not perfect, I do believe they are useful. I believe they will give you a starting point that will improve your ability to refactor efficiently. That is what they do for me.

Extract Method (110)

Turn the fragment you are looking on into a method which explains by its name the purpose of the method.

Motivation: To have short, well-named methods so other methods can use it.

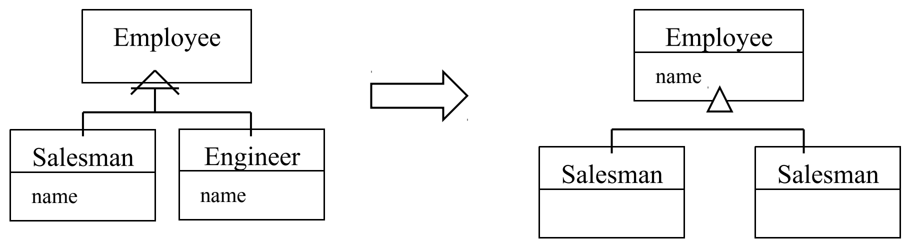
Before	**After**

`void f() {`
`    …`
`    // compute the score`
`    score = a * b * c;`
`    score -= discount;`
`}`
	`void f() {`
	`        …`
	`        computeScore();`
	`}`
	
	`void computeScore() {`
	`        score = a * b * c;`
	`        score -= discount;`
	`}`

Pull Up Field

Move the field to the superclass

Motivation: To reduce duplication in two ways. It removes the duplicate data declaration and allows you to move from the subclasses to the superclass behavior that uses the field.



Substitute Algorithm(139)

Replace the body of the method with the new algorithm.

Motivation: If you find a clearer way to solve a problem, remove the whole algorithm and replace it with the clearer one.

Before

```
String foundPerson(String[] people) {
        for (int i=0; i < people.length; i++) {
                if(people[i].equals("Magnus")) {
                        return "Magnus";
                }
                if(people[i].equals("Erik")) {
                        return "Erik";
                }
                if(people[i].equals("Brad")) {
                        return "Brad";
                }
        }
        return "";
}
```

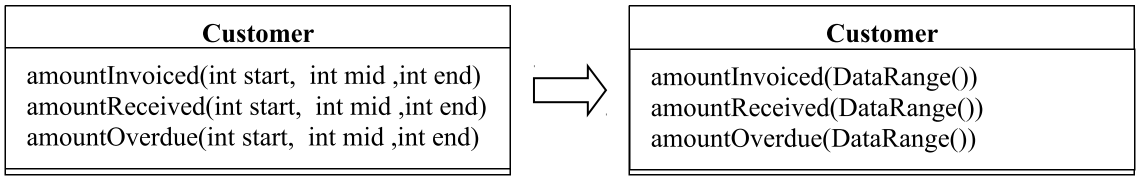
After

```
String foundPerson(String[] people) {
        List candidates = Arrays.asList(new String[] {"Magnus", "Erik", "Brad" });
        for (int i=0; i < people.length; i++) {
                if(candidates.contains(people[i]))
                        return people[i];
                return "";
        }
}
```

Introduce Parameter Object (295)

Replace parameters with an object.

Motivation: By grouping the data together to an object, reduces the size of the parameter Lists, and long parameter lists are hard to comprehend.


Customer
amountInvoiced(int start, int mid ,int end)
amountReceived(int start, int mid ,int end)
amountOverdue(int start, int mid ,int end)
Customer
amountInvoiced(DataRange())
amountReceived(DataRange())
amountOverdue(DataRange())

Replace Temp With Query (120)

Extract expression into a method, then replace all references to the temp with the expression.

Motivation: By replacing the temp with a query method, any method in the class can now get hold of the information. It is also easier to do *Exctract Method* since one has less variables to deal with.

Before

```
double basePrice = quantity * itemPrice;
if(basePrice > 1000)
        return basePrice * 0.95;
else
        return basePrice * 0.98;
```

After

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

Preserve Whole Object (288)

Send the whole object instead

Motivation: To make the parameter list more robust and consistent. You also move more behavior when having access to the object.

Before

```
int low       = daysTempRange().getLow();
int hight     = daysTempRange().getHigh();
withinPlan = plan.withinRange(low. Range);
```

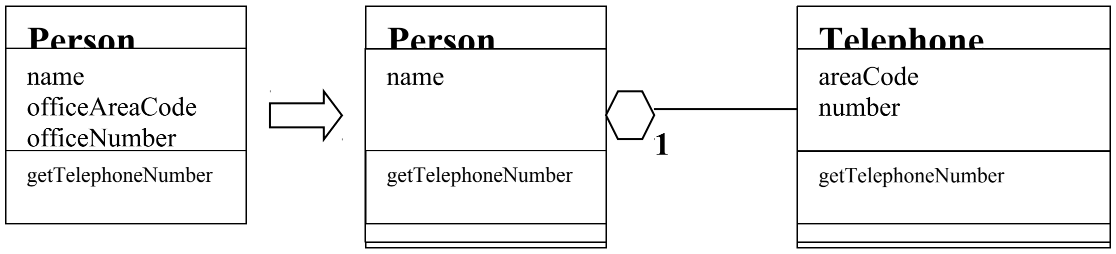
After

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

Extract Class (149)

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

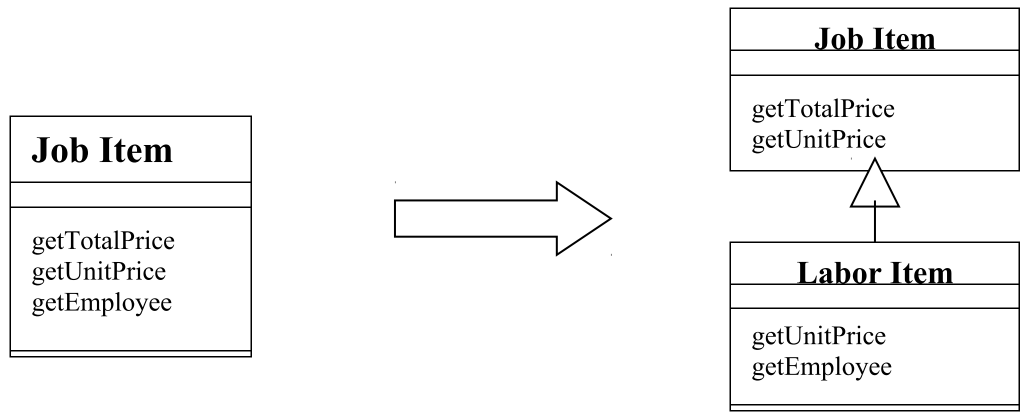
Motivation:As for *Extract Method*, extract class if it gets too big and hard to understand.



Extract Subclass(330)

Create a subclass for that subset of features.

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



Replace Parameter With Method(292)

Remove the parameter and let the receiver invoke the method.

Motivation: If a method can get a value that is passed in as parameter by another means, that is preferred.

Before

```
int basePrice = quantity * itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice,discountLevel);
```

After

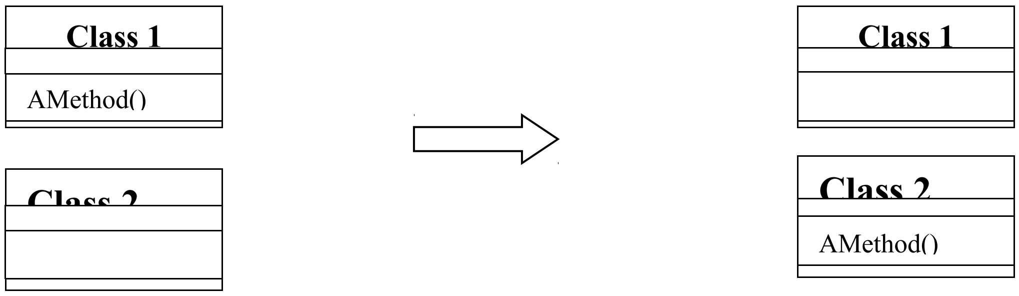
```
// Removed the parameter and let the receiver invoke the getDiscountLevel method

int basePrice = quantitiy * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Move Method(142)

Create a new method with a similar body in the class it uses most.

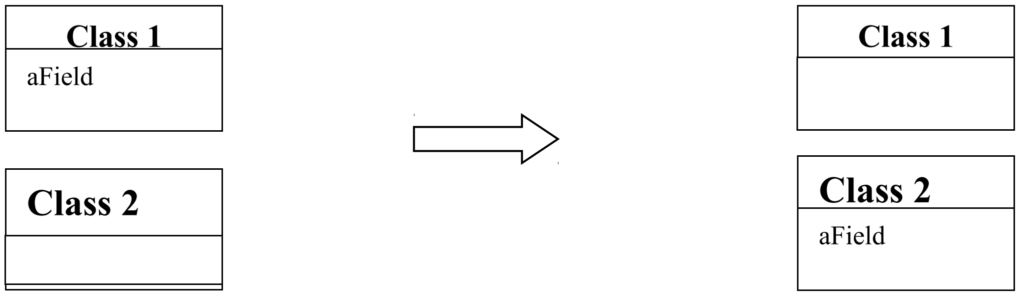
Motivation: Move the method when it is used by more features of another class than the class on which it is defined.



Move Field(146)

Create a new field in the target class, and change all its users.

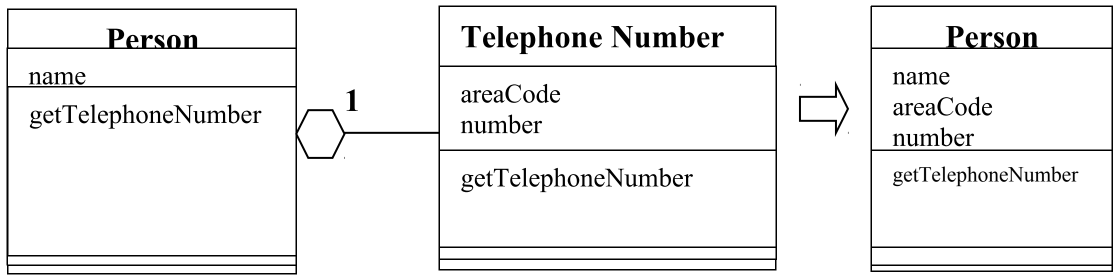
Motivation: Move the field when it is used by another class more than the class on which it is defined.



Inline Class(154)

Move all its features into another class and delete it

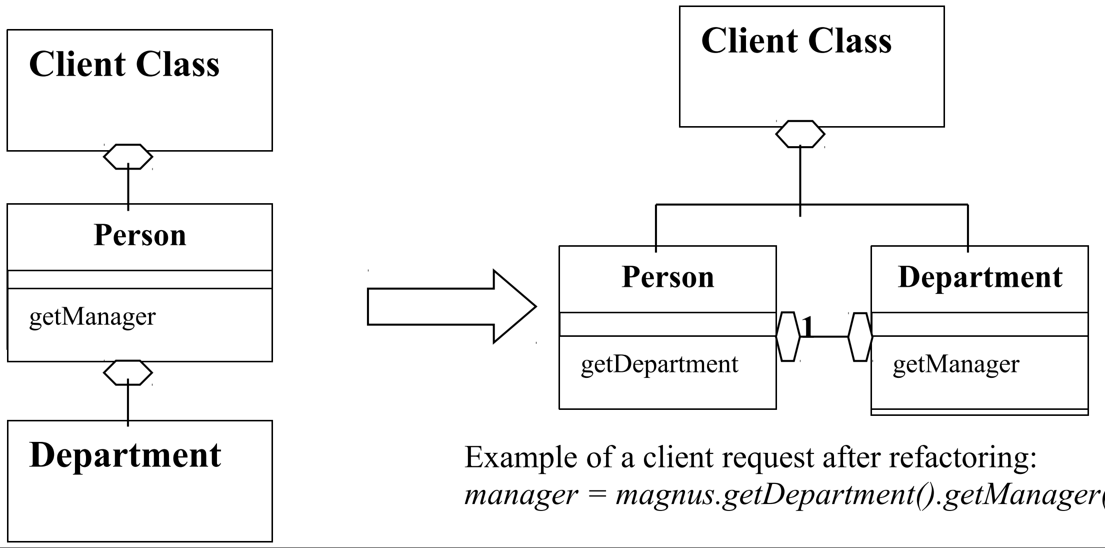
Motivation: Used when the class isn't doing very much.



Remove Middle Man(160)

Get the client to call the delegate directly

Motivation: A class is doing to much simple delegation.

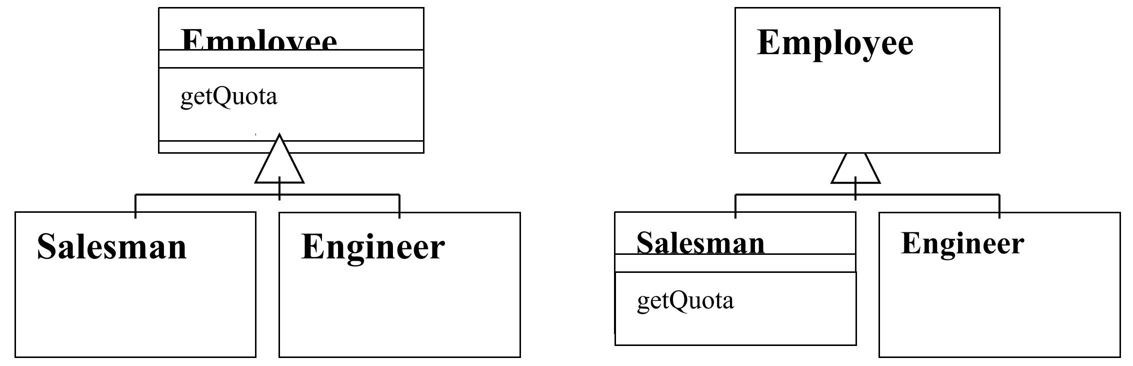


Example of a client request after refactoring:
manager = magnus.getDepartment().getManager();

Push Down Method (328)

Move the Method to the meaningful subclass

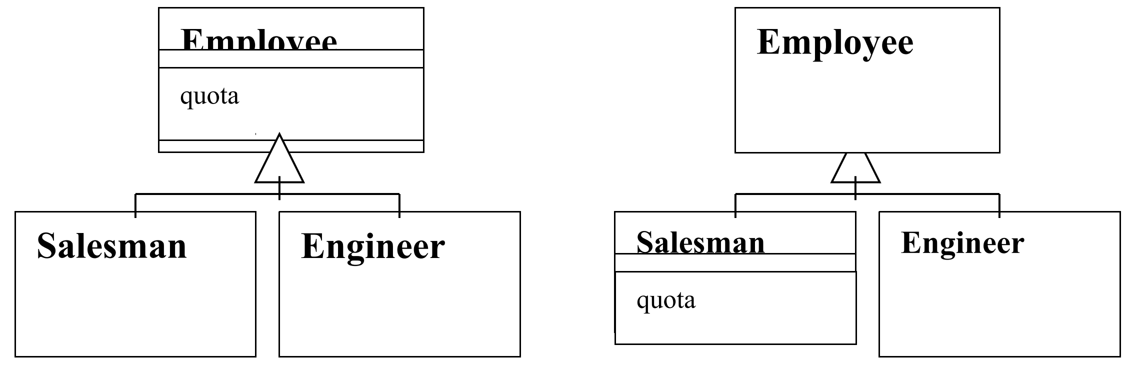
Motivation: Behavior on a superclass is relevant only for some of its subclasses.



Push Down Field (329)

Move the field to a meaningful subclass

Motivation: A field is used only by some subclasses.


Employee
quota
Salesman
Engineer
Employee
Salesman
quota
Engineer