

Object Oriented Design Patterns - I

Agile Programming

Giuseppe Lipari
<http://retis.sssup.it/~lipari>

Scuola Superiore Sant'Anna – Pisa

November 28, 2011

Control variables

- The software development process can be controlled with four different variables
 - Cost
 - Time
 - Quality
 - Scope
- A software development game:
 - External forces (customers, managers) can pick the value of 3 out of 4 variables
 - The goal is to the software development process to be successful (i.e. develop the software)
- Some managers think they can control all of the four variables!

- *Cost*: more money can grease the skids a little, too much money too soon creates more problems than it solves. Too little money will make the project fail
- *Time*: more time to deliver can improve the quality and increase scope. Too much time will hurt, because the best feedback is from the real product in the market
- *Quality*: Difficult to control. You can sacrifice quality and gain a little in the very short term, but also getting larger problems in the long term
- *Scope*: Less scope makes it possible to deliver better quality, develop sooner and/or cheaper

Relationships between variables

- There is not an easy relationship between the variables
 - Often you cannot speed up a project by spending more money. For example, hiring more programmers can hurt because you need more coordination, the new ones have to learn, etc.
 - Cost is the most constrained variable. The investment has to start small and grow over time
 - Constraints on cost make the manager crazy. They reason on an annual budget, on constraints from upper layers managers, etc. Managers drive everything from cost.
 - Cost has to do with prestige (the higher the cost of the project, the bigger is the manager)
 - Time is also often an external variable (market pressure, etc.)
 - Quality is a strange variable. Often by insisting on quality from the beginning you get *more projects* in a *smaller* time.
- *Scope* is the only one variable that can be controlled easily
 - Less scope means: less time to deliver something meaningful; less cost; more control on timing; opportunity to achieve a higher quality

Cost of change

If it works, don't fix it

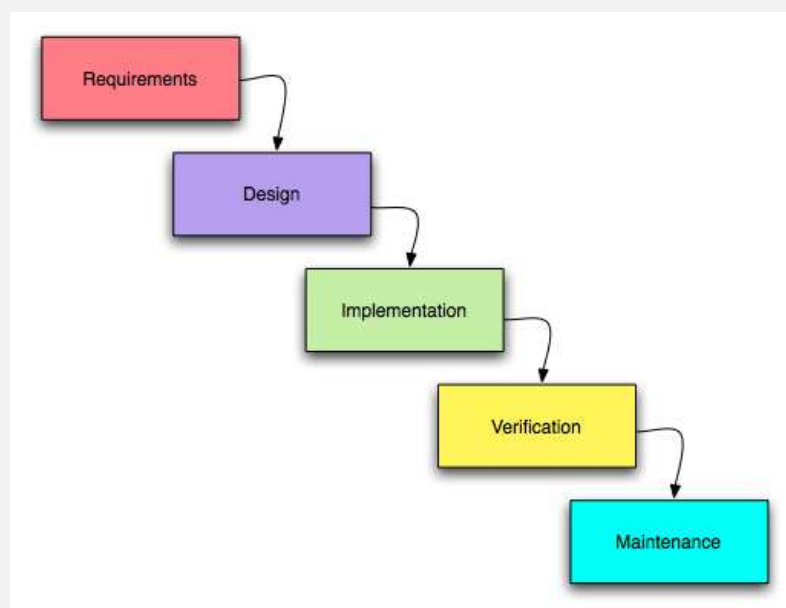
- This is an old engineers' saying, very well known and respected by everybody
- It is strictly related to Murphy's Law

Anything that can go wrong, will go wrong

- It makes people scared about changing existing things
- It is also related to the *fear of change* and the *raising cost of change*

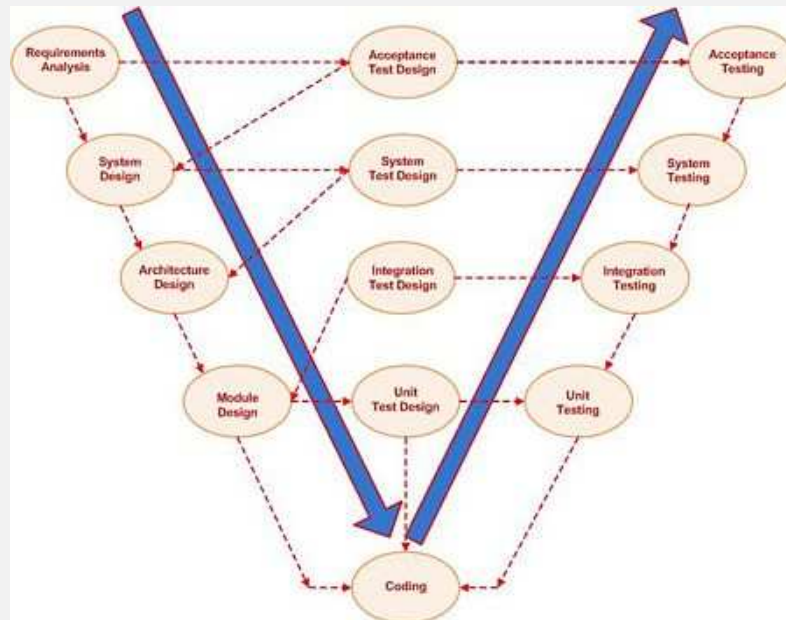
The waterfall model

- A typical software development model is the “waterfall”
 - Start from the specification, ends with deployment
 - Still followed, especially in safety critical



V-Model

- A modern version of the waterfall model, with emphasis on integration testing and verification
- again, used in safety-critical and mission-critical systems



Development cost

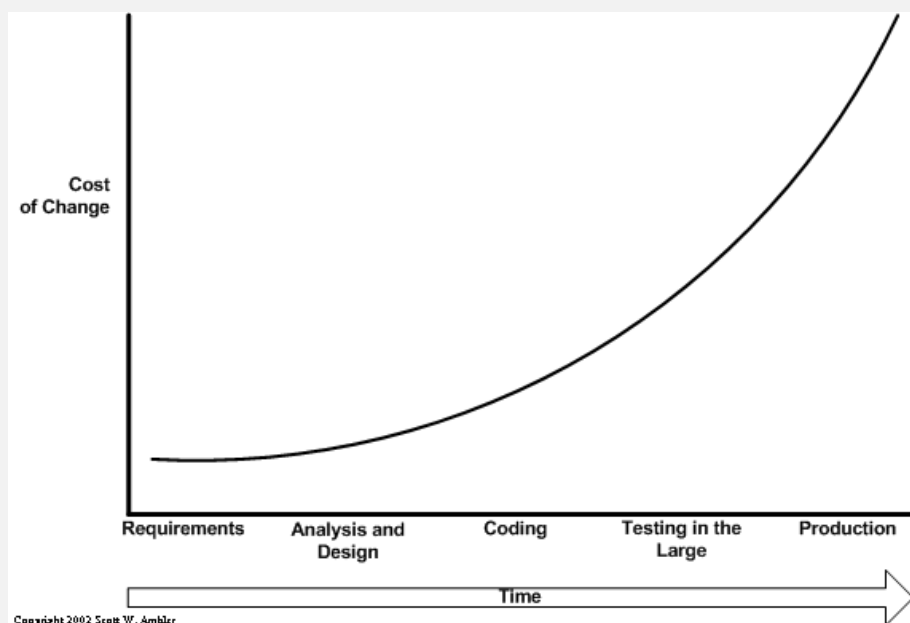
- In the previous models, a lot of emphasis is put on early phases of development, and in particular: requirements, specification and design
- Requirements: what the client desires
 - Requirement Analysis is critical for the success of a software project!
- Specification: more formal and detailed version of the requirements, often for internal use
 - They are also used to contain technical details, estimate efforts clearly specify input/output relationship, etc.
 - Can be used to derive tests
 - In safety-critical software, this phase is considered essential and critical for the success of the software!
- Design: breaks up the project into subsystems, their interactions, etc.
 - This is considered very important, since a wrong design can compromise the final product functionality, performance, etc.

Everything is critical

- As you can see, without “feedback” everything must be known and precisely specified since the beginning
 - Since everybody only looks forward to the next phase, any error requires changing a lot of things
 - therefore, there is a lot of emphasis on making them right since the beginning
- of course, this is not how it works in practice
 - Requirements are always imprecise, often volatile, sometime in conflict between them
 - specifications are incomplete, not-precise, or even not existing
 - errors do happen in the early phases of the design, and hence it is necessary to “patch” the design, the code and the tests quite often
 - errors are in human nature

The raising cost of change

- In the two models, we can draw the following curve to describe the cost of changing anything in software



Fear of change

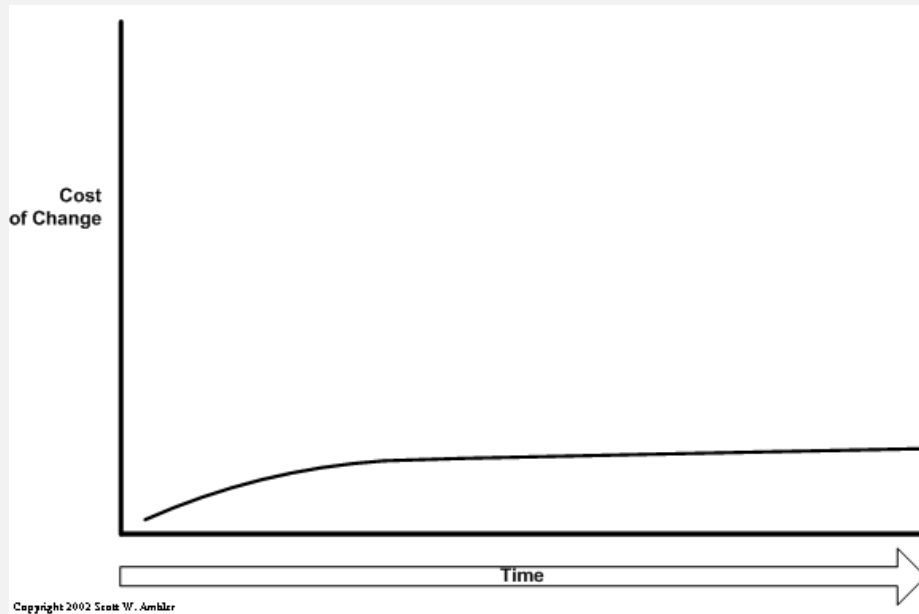
- Therefore, there is a *fear to change* anything
 - Especially if it works!
- At the same time, changes happens naturally
 - The hardware is not adequate
 - The design is not flexible
 - A new requirements is suddenly added
 - Some module will not be ready on time
 - etc.
- Lot of stress!

Code degradation

- Changing something in the design is difficult, expensive, and highly feared when we are in the coding stage
 - therefore, problems in requirements, specification and design are solved with modifications in the code (patches, hacks, etc.)
 - such patches start accumulating little by little
 - What started as a clean design ends up as unclean and messy code
- Every programmer can tell that there is a difference between clean and dirty code, and can see such difference when she/he sees it
- dirty code causes stress
- clean code is easier to maintain, debug, extend
- in a perfect world, we will always have to deal with clean code

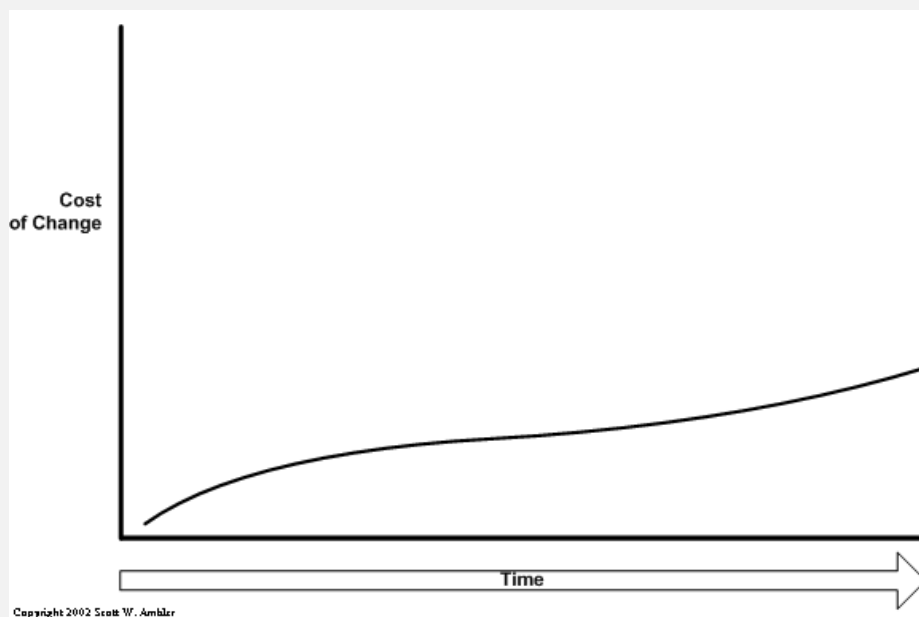
A dream

- What if the cost of change remains constant across the software development life cycle?



More realistic

- The previous figure is just a dream, but maybe with appropriate techniques we can achieve this:



Keep the code clean

- If we could continuously maintain the code, and keep it clean and as simple as possible, it will be easier to change it
- It requires a method:
 - The code cleaning should become routine and be integrated into the process
- Is it worth it?
 - It depends on how many changes you anticipate in your process
 - if they are many, it means you will need to go back and change your code quite often
 - better to keep it clean

Telling the story

- Once upon a time, a consultant made a visit to a development project
- During analysis, he discovered that the hierarchy of classes was rather messy and could be improved
- He recommended the project management to spend some time cleaning up the code
- Of course, they were not happy
 - The code works, why you should modify it?
 - also, we are behind! There is still a lot to do! We are under time and cost budget
- The consultant convinced the programmers of the problems, and they spend a couple of days cleaning the code
- the amount of code was *reduced* significantly, and was clearer
- The managers were not happy
 - Who care about nice code? The client does not care, so we do not care
- The managers sent the consultant away

- The consultant was Martin Fowler (author of the book: “Refactoring”)
- Six months later the project was cancelled because it was too late
- Another consultant (Kent Beck, author of “Extreme Programming”) was hired, and started rewriting the code almost from scratch
- The reason of the failure was that the code was too heavy and too complex to debug and maintain
- (This is the preface of “Refactoring”)

What is refactoring?

- It is the process of changing a software system in such a way that it does not modify the external behaviour, but improving its internal structure
- It is possible to improve the design of existing software *after* the software has been written?
- Of course, the answer is “probably yes”
- However, to do this we must be careful to do it following a rigorous discipline:
 - Do it in little incremental steps
 - At every step, run all tests to see if the code is still doing what we expect it to do
 - Therefore, you will not introduce new bugs
- Step by step, little by little, the design will improve

Why?

- Is beauty important?
 - No, the things that matter are about functionality and performance
- However, beauty is not independent from other nice properties
 - Simplicity is beautiful
 - Independence and isolation are beautiful
 - Flexibility is beautiful
 - etc.
- In other words, nice code is also code that has a lot of useful properties that will make our life easier later on

Refactoring and design

- in classical software development, you do a very careful up-front design at the beginning
- design is followed by coding
- the emphasis is on making **THE** design
 - The one that is *correct* and will solve all present and future problems
 - since it is trying to solve **ALL** future problems, extensions, desires, etc., it is often overly complex
 - it ends up that it will contain many features that will never be implemented because they are not needed
- With refactoring there is less emphasis on up-front design
 - You start with one *reasonable design*
 - You don't try to solve all future problems, but just the ones that you foresee now
 - therefore, the design becomes simple
 - if an extension is needed, you refactor the design *when you need it*

When not to refactor

- Remember: code has to work most correctly before you can re-factor
- If there are bugs:
 - first debug
 - then write a test
 - repeat until problems are solved
- once all tests run smoothly, you can re-factor
- So, don't refactor if you have too many bugs
 - Maybe you should consider rewriting from scratch!
- Don't refactor when you are close to a deadline
 - refactoring pays off in the long term, after the deadline
 - if you are in a hurry, refactor later

Where to start

- Every time we need to refactor, the first thing is to build the tests that check what is going on
 - Of course, it is much better if the tests are already there
 - It is the only way to understand if the change is correctness
 - also, it builds confidence into the programmer
 - it makes the fear of change go away
- let's talk about test

Refactoring methods

- The book lists **72** refactoring methods
 - Each method is described with a couple of pages, containing a small example
- Methods are organized into groups
 - **Composing methods**: how to organize the code into methods
 - Extract / Inline Method, Replace Temp with Query, Split Temporary Variable, etc.
 - **Move Features Between Objects**: how to move code
 - Move Method, Move Field, Extract / Inline Class, etc.
 - **Organize Data**: how to organize data into classes
 - Replace Data Value With Object, Self Encapsulate Field, etc.
 - **Simplifying Conditional Expression**
 - Decompose Conditional, Replace Conditional With Polymorphism, etc.
 - **Making Method Calls Simpler**
 - Add/Remove Parameter, Rename Method, Parameter Object, etc.
 - **Dealing with Generalization**
 - Replace Inheritance with Delegation, (and viceversa), etc.

Bad smell

- Suppose you are looking at some code
- Should I refactor or not?
- In the book, the authors say that code should be refactored when it “smells”
 - **Duplicated Code**
 - **Long Method**
 - **Switch Statement**
 - **Data Class**
 - **Shotgun Surgery**
 - **Long Parameter List**
 - **Alternative Classes with Different Interfaces**

The problem with duplicated code

- “Number one in the stink parade”
- When you see the same “code structure” in different places in the code
- The idea is to generalize the code, and put it into a separate method
- When the code is exactly the same, it is easy:
 - Use Extract Method, then call the method from all the places
- If the code is in two classes derived from the same base class
 - Use Extract Method and Pull-Up Field
- If the similar code is in two unrelated classes
 - you can consider Extract Class

Extract Method

- This is one of the simplest method
 - Also contained in Eclipse CDT

```
void printAll(double amount) {  
    printBanner();  
  
    System.out.println("Name    : " + _name);  
    System.out.println("Amount  : " + amount);  
}
```



```
void printDetails(double amount) {  
    System.out.println("Name    : " + _name);  
    System.out.println("Amount  : " + amount);  
}  
void printAll(double amount) {  
    printBanner();  
    printDetails(amount);  
}
```

Description

- When to use it
 - in case of duplicated code
 - in case of a method that is too long,
- Small methods are more readable and easier to debug and understand
- Pay attention to names
- The reverse of this method is called Inline Method
 - Use when a function is too silly

Mechanics

- 1 Create a new method (target method), and name it
- 2 Copy the code to be extracted into the target method
- 3 Scan the extracted code for reference to any variable that is local to the source method
 - Local temporary variables that are used only in the scope of the extracted code become local variables in the new method
 - Local variable that are used both in the source method and in the extracted code become parameters of the method
 - they can be passed by value or by reference, depending on their use in the two methods
 - Try to eliminate (or minimize) parameters passed by reference, either by using an appropriate return value, or by splitting the code in a different way
- 4 Compile
- 5 Delete the extracted code from the source method, and replace it with a method call
- 6 Compile and test

Example

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Example - solved

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Replace a Method with an Object

- This is useful when you have a very long and complex method that uses a lot of local variables
- Also, the local variables are used in a complex way, so that it is difficult or impossible to use Extract Method

```
class Order {  
    ...  
    double price() {  
        double primary;  
        double secondary;  
        double numElements;  
        ...  
    }  
};
```

```
class PriceCalc {  
    double primary;  
    double secondary;  
    double numElements;  
    ...  
    void compute() {...}  
};  
  
class Order {  
    ...  
    double price() {  
        PriceCals pc(this);  
        return pc.compute();  
    }  
};
```

Mechanics

- 1 Create a new class, name it after the method
 - 2 Give the new class one field pointing to the original object
 - So that you can call methods on it
 - 3 Give the new class one field for every local variable in the source method, and for every parameters
 - 4 Create the constructor to take all parameters, and fill up the fields
 - 5 create a method called compute(), and copy all the source method into compute
 - 6 compile
 - 7 replace the source method code with the creation of the new object, and a call to compute()
 - 8 compile and test it
- Now, you can refactor the long compute () method as you wish with Extract Method, because all local variables are now field
 - once it is simplified, you may discover that the object is no longer necessary ...

Pull-up field

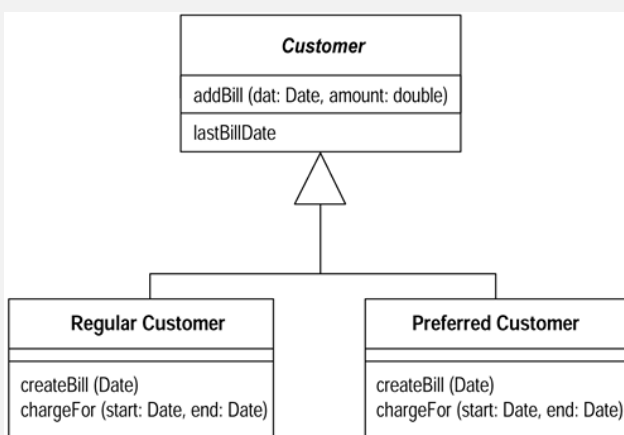
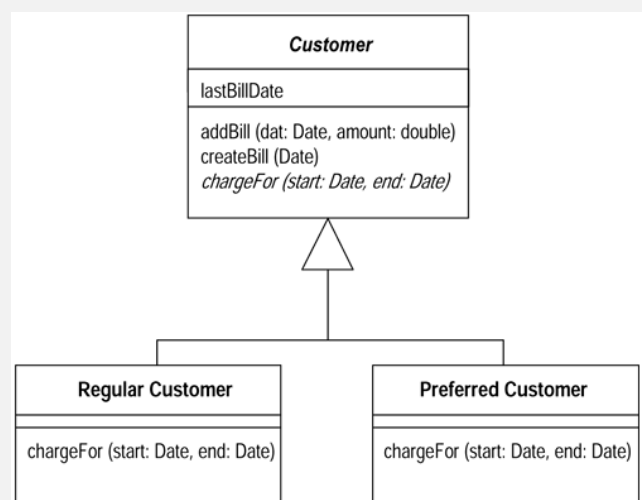
- When you have a field that is duplicated in two sibling classes
 - It may or may not have the same name: be careful to understand if it is actually the same field!
- Pull-up field is used to move it in the base class
 - if they have different names, first rename, compile and test
 - then move the field up, compile and test

Pull-up Method

- You can do the same with method, but you have to be even more careful
 - You should check if the two methods are actually doing the same thing
 - If they are not identical, but just similar, consider to use Extract Method on both to generalize a common “core” that you will later pull-up
 - One problem is when the methods refer other methods that are in the derived classes
 - in such a case, you may consider defining virtual methods in the base class, so that the call is redirected appropriately

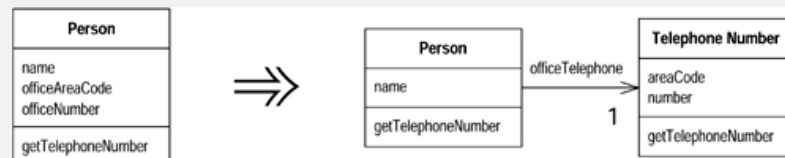
- ❶ Inspect the methods to ensure they are identical
 - if not, consider first some other refactoring to make them identical
- ❷ If they have different signature, make them uniform
- ❸ Create a method in the base class, copy the code, compile and test
- ❹ If everything works, delete one subclass method, compile and test
- ❺ continue to delete methods in the subclasses one at time, compile and test
- ❻ Search where the method is called, and if done via a pointer, see if you can change the pointer type to the base class

Example



Extract Class

- You have one class that does the work that should be done by two
- For example when you have a class with a lot of methods and data fields
 - Often, the class is too big to understand easily what is going on



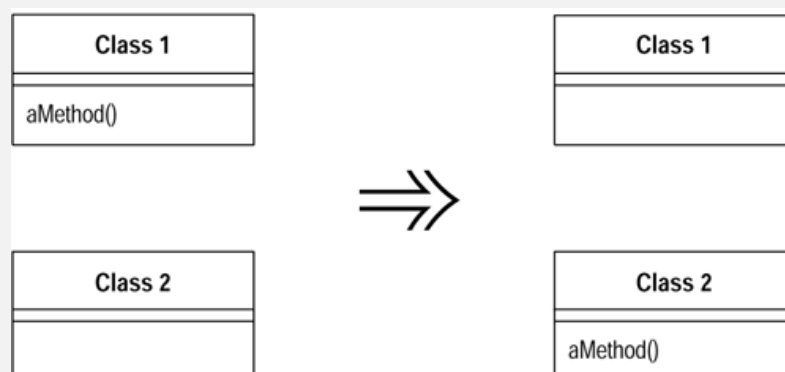
Mechanics

- 1 Decide how to split the responsibilities of the class.
- 2 Create a new class to express the split-off responsibilities.
 - If the responsibilities of the old class no longer match its name, rename the old class
- 3 Make a link from the old to the new class.
 - You may need a two-way link. But don't make the back link until you find you need it
- 4 Use Move Field on each field you wish to move.
- 5 Compile and test after each move.
- 6 Use Move Method to move methods over from old to new. Start with lower-level methods (called rather than calling) and build to the higher level.
- 7 Compile and test after each move.

- 8 Review and reduce the interfaces of each class.
 - If you did have a two-way link, examine to see whether it can be made one way.
- 9 Decide whether to expose the new class. If you do expose the class, decide whether to expose it as a reference object or as an immutable value object

Move Method

- A method is, or will be, using or used by more features of another class than the class on which it is defined
- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



- Moving methods is the bread and butter of refactoring.
- move methods when
 - classes have too much behavior
 - classes are collaborating too much and are too highly coupled
- By moving methods around, classes become simpler and they end up being a more crisp implementation of a set of responsibilities

Mechanics

- 1 Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.
 - If a feature is used only by the method you are about to move, you might as well move it, too.
 - If the feature is used by other methods, consider moving them as well.
 - Sometimes it is easier to move a clutch of methods than to move them one at a time.
- 2 Check the sub- and superclasses of the source class for other declarations of the method.
 - If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target.
- 3 Declare the method in the target class.
 - You may choose to use a different name, one that makes more sense in the target class.

- 4 Copy the code from the source method to the target. Adjust the method to make it work in its new home.
 - If the method uses its source, you need to determine how to reference the source object from the target method.
 - If there is no mechanism in the target class, pass the source object reference to the new method as a parameter.
 - If the method includes exception handlers, decide which class should logically handle the exception.
 - If the source class should be responsible, leave the handlers behind.
- 5 Compile the target class.

- 6 Determine how to reference the correct target object from the source.
 - There may be an existing field or method that will give you the target.
 - If not, see whether you can easily create a method that will do so.
 - Failing that, you need to create a new field in the source that can store the target.
 - This may be a permanent change, but you can also make it temporarily until you have refactored enough to remove it.
- 7 Turn the source method into a delegating method.
- 8 Compile and test.
- 9 Decide whether to remove the source method or retain it as a delegating method.
 - Leaving the source as a delegating method is easier if you have many references.

- 10 If you remove the source method, replace all the references with references to the target method.
 - You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace.
- 11 Compile and test.

Example

```
class Account...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn > 7)
                result += (_daysOverdrawn - 7) * 0.85;
            return result;
        }
        else return _daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result += overdraftCharge();
        return result;
    }
    AccountType _type;
    int _daysOverdrawn;
```

- let's move the overdraft charge method over to the account type

How to transform

- copy the method body over to the account type and get it to fit.

```
class AccountType...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7)
                result += (daysOverdrawn - 7) * 0.85;
            return result;
        }
        else return daysOverdrawn * 1.75;
    }
}
```

Transformation

- replace the source method body with a simple delegation:

```
class Account...
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }
}
```

```
class Account...
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result +=
            _type.overdraftCharge(_daysOverdrawn);
        return result;
    }
}
```

Last step

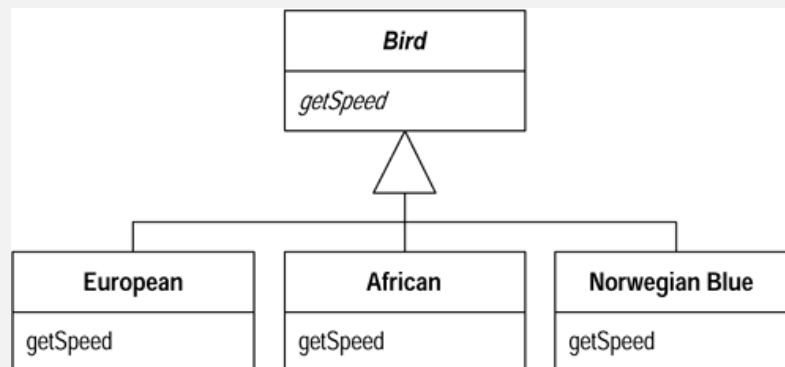
- I may need to pass a reference to the original class instead of a single parameter

```
class AccountType...
    double overdraftCharge(Account &account) {
        if (isPremium()) {
            double result = 10;
            if (account.getDaysOverdrawn() > 7)
                result += (account.getDaysOverdrawn() - 7) * 0.85;
            return result;
        }
        else return account.getDaysOverdrawn() * 1.75;
    }
}
```

Replace conditional with polymorphism

- A conditional chooses different behavior depending on the type of an object.
- Example: compute the speed of a Bird

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw RuntimeException("Should be unreachable");
}
```



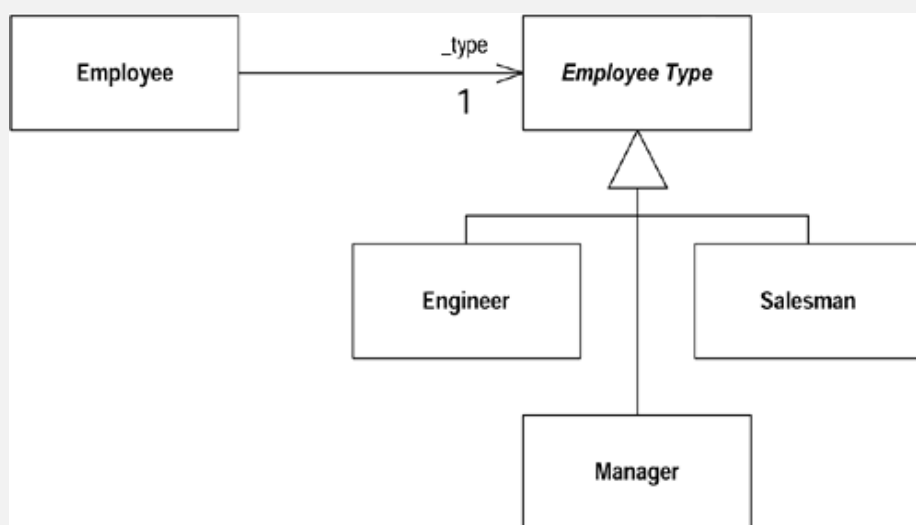
Mechanics

- Before you can begin with Replace Conditional with Polymorphism you need to have the necessary inheritance structure.
 - You may already have this structure from previous refactorings.
 - If you don't have the structure, you need to create it.
- 1 If the conditional statement is one part of a larger method, take apart the conditional statement and use Extract Method.
 - 2 If necessary use Move Method to place the conditional at the top of the inheritance structure.
 - 3 Pick one of the subclasses. Create a subclass method that overrides the conditional statement method. Copy the body of that leg of the conditional statement into the subclass method and adjust it to fit.
 - You may need to make some private members of the superclass protected in order to do this.

- 4 Compile and test
- 5 Remove the copied leg of the conditional statement.
- 6 Compile and test.
- 7 Repeat with each leg of the conditional statement until all legs are turned into subclass methods.
- 8 Make the superclass method abstract.

Example

- The inheritance structure



The switch code

```
class EmployeeType...
    int payAmount(Employee &emp) {
        switch (getTypeCode()) {
            case ENGINEER:
                return emp.getMonthlySalary();
            case SALESMAN:
                return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
                return emp.getMonthlySalary() + emp.getBonus();
            default:
                throw RuntimeException("Incorrect Employee");
        }
    }
}
```

Example - cont

- I change the payAmount method in Employee to delegate to the new class:

```
class Employee...
    int payAmount() {
        return _type.payAmount(this);
    }
```

- First, copy the Engineer leg of the case statement onto the Engineer class.

```
class Engineer...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary();
    }
```

- Carry on until all methods are implemented

```
class Salesman...  
    int payAmount(Employee emp) {  
        return emp.getMonthlySalary() + emp.getCommission();  
    }  
  
class Manager...  
    int payAmount(Employee emp) {  
        return emp.getMonthlySalary() + emp.getBonus();  
    }
```

- and then declare the superclass method abstract:

```
class EmployeeType...  
    abstract int payAmount(Employee emp);
```