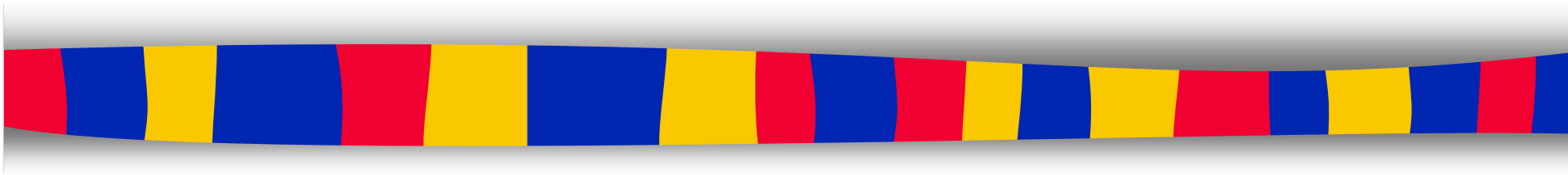
EECS 4314

Advanced Software Engineering



Topic 10:

Software Refactoring

Zhen Ming (Jack) Jiang

Acknowledgement

- Some slides are adapted from Professor Marty Stepp, Professor Oscar Nierstrasz

Relevant Readings

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring – Improving the Design of Existing Code.
- Steve McConnel. Code Complete: : A Practical Handbook of Software Construction. (Chapter 24)

Problem: "Bit rot"

- After several months and new versions, many codebases reach one of the following states:
  - *rewritten* : Nothing remains from the original code.
  - *abandoned* : Original code is thrown out, rewritten from scratch.
- Why?
  - Systems evolve to meet new needs and add new features
  - If the structure of the code does not also evolve, it will "rot"
  - This can happen even if the code was initially reviewed and well-designed at the time of check-in

Software maintenance

- **Software maintenance**: Modification or repair of a software product after it has been delivered.
- Purposes:
  - fix bugs
  - improve performance
  - improve design
  - add features,
  - etc.
- Studies have shown that ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).

Maintenance is hard

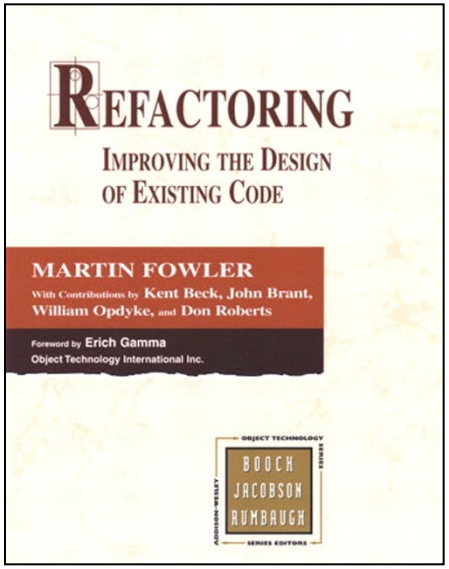
- It's harder to maintain code than write your own new code.
  - must understand code written by another developer, or code you wrote at a different time with a different mindset
  - most developers dislike software maintenance
- Maintenance is how developers spend much of their time.
- It pays to design software well and plan ahead so that later maintenance will be less painful.
  - Capacity for future change must be anticipated

Refactoring

- **Software refactoring** is the systematic practice of improving application code's structure without altering its behavior.
  - Incurs a short-term time/work cost to reap long-term benefits
  - A long-term investment in the overall quality of your system.
- Refactoring is not the same thing as:
  - adding features
  - debugging code
  - rewriting code

A Brief History of Code Refactoring

- Invented by two computer science graduate students in the late 1980s:
  - Bill Opdyke at University of Illinois at Urbana-Champaign
  - Bill Griswold at University of Washington
- Canonical reference



Why refactor?

- Why fix a part of your system that isn't broken?
  - Each part of your system's code has the following three purposes. If the code does not do one or more of these, it is "broken."
    1. to execute its functionality,
    2. to allow change,
    3. to communicate well to developers who read it.
- Refactoring:
  - changes internal structure of the program and improves software's design
  - makes it easier to understand and cheaper to modify
  - do not change its observable behaviour

When to refactor?

- When is it best for a team to refactor their code?
  - best done **continuously** (like testing) as part of the process
  - hard to do well late in a project (like testing)
- Refactor when you identify an area of your system that:
  - isn't well designed
  - isn't thoroughly tested, but seems to work so far
  - now needs new features to be added

Bad smells

If it stinks,
change it

Kent Beck grandma
discussing child-rearing philosophy

Signs you should refactor - “Code Smells”

- code is **duplicated**
- a routine is **too long**
- a loop is too long or **deeply nested**
- a class has poor **cohesion**
- a class uses too much **coupling**
- inconsistent level of **abstraction**
- too many **parameters**
- to **compartmentalize** changes (change one place → must change others)
- to modify an **inheritance hierarchy** in parallel
- to **group related data** into a class
- a **"middle man"** object doesn't do much
- **poor encapsulation** of data that should be private
- a **weak subclass** doesn't use its inherited functionality
- a class contains **unused code**

Code Smells

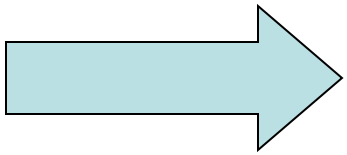
Code Smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
  - (change one place → must change others)
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
  - (subclass doesn't use inherited members much)
- Comments

What smells in here?

```
void funcA()
{
    int x, y = 2;
    x = y * y;
    printf("%d", x);
}

void funcB()
{
    int x, y = 4;
    x = y * y;
    funcC(x);
}
```



Duplicate Code

```
void funcA()
{
    …
    x = sqr(y);
    …
}
void funcB()
{
    …
    x = sqr(y);
    …
}
int sqr(int x)
{
    return x * x;
}
```

Extract Method

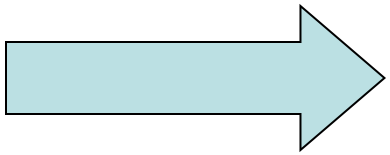
Duplicated Code

- Code repeated in multiple places
- Refactoring
  - Extract Method
  - Extract Class
  - Pull Up Method
  - Form Template Method

What smells in here?

```
void funcA(
   int param1,
   int param2,
   char* param3,
   float param4,
   float param5,
   void* param6)
{
int temp1, temp2,
   temp3;

// Do stuff with all
// this data
}
```



Long Parameter List

```
class newObj
{
public:
    int getParam1();
    int getParam2();
    char* getParam3();
    float getParam4();
    float getParam5();
    void* getParam6();
}
void funcA(newObj obj)
{
int temp1, temp2, temp3;
// Do stuff with all
// this data
}
```

Introduce Parameter Object

Long Parameter List

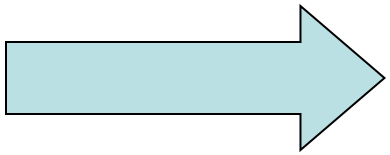
- Method needs too much external information. It becomes hard to understand and inconsistent
- Refactoring
  - replace parameter with method (receiver explicitly asks sender for data via sender getter method)
  - replace parameters with a member field in a dedicated object

What smells in here?

```
void funcA(acctObj acct)
{
int payment;

payment =
    acct.getBalance();
payment /=
    acct.getTerm();
payment *=
    acct.getRate();

// More stuff with
// payment
}
```



Feature Envy

```
void funcA(acctObj acct)
{
int payment;

payment =
    acct.getPayment();

// More stuff with
// payment
}
Class acctObj
{
…
int getPayment(); //new!
…
}
```

Extract Method -> Move Method

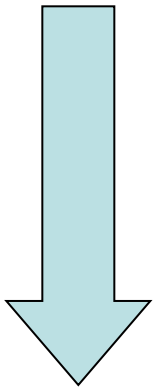
Feature Envy

- Method in one class uses lots of pieces from another class. This method needs too much information from another object
- Refactoring
  - move method to the other class

What smells in here?

```
class Animal {
   final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
   int myKind;  // set in constructor
   ...
   String getSkin() {
      switch (myKind) {
         case MAMMAL: return "hair";
         case BIRD: return "feathers";
         case REPTILE: return "scales";
         default: return “skin";
      }
   }
}
```

Switch Statements



switch statements (improved)

```
class Animal {
    String getSkin() { return “skin"; }
}
class Mammal extends Animal {
    String getSkin() { return "hair"; }
}
class Bird extends Animal {
    String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
    String getSkin() { return "scales"; }
}
```

switch statements

- switch statements are very rare in properly designed object-oriented code
  - Therefore, a switch statement is a simple and easily detected “bad smell”
  - Of course, not all uses of switch are bad
  - A switch statement should *not* be used to distinguish between various kinds of object
- There are several well-defined refactorings for this case
  - The simplest is the creation of subclasses

How is this an improvement?

- Adding a new animal type, such as Amphibian, does not require revising and recompiling existing code
- Mammals, birds, and reptiles are likely to differ in other ways, and we've already separated them out (so we won't need more switch statements)
- We've gotten rid of the flags we needed to tell one kind of animal from another
- We're now using Objects the way they were meant to be used

More “smelly” code

- **Divergent Change**: when a class is changed in different ways for different reasons. E.g., “I have to change these three methods every time I get a new database; I have to change these four methods every time there is a new functional instrument”.
  - *Extract Class: Separate out the varying code into varying classes that either subclass or are contained by the non-varying class*
  - *Use Visitor or Self Delegation patterns*
- **Shotgun Surgery**: [opposite of the divergent change] to accommodate a change to this class (i.e., to one of its fields/methods) triggers lots of small changes to several other classes
  - *Move Method*
  - *Move Field*
  - *Inline Class*

More “smelly” code (2)

- **Parallel Inheritance Hierarchies**: special case of shotgun surgery where subclass of one class requires subclass of another class every time. This could be symptom of something wrong in the class hierarchy that can be corrected by redistributing responsibilities among the classes
  - Extract subclass
- **Data Clumps**: Data always used together (x,y -> point);
  - *Extract Class (*move these to a new class)
  - *Introduce Parameter Object*
- **Primitive Obsession**: group related primitives in a function into a new class
  - *Replace Data Value with Object,*
  - *Replace Type Code with Class*

Even more “smelly” code

- Blog: a class implementing several responsibilities, having a large number of attributes, operations, and dependencies with data classes.
  - *Extract Class*
- Lazy Class: classes enjoy doing nothing if given the chance; must pull its own weight
  - *Collapse Hierarchy,*
  - *Inline Class*
- Speculative Generality: somebody over-engineered or over-designed -> too complicated
  - *Collapse Hierarchy,*
  - *Inline Class,*
  - *Remove Parameter*
- Temporary Field: a member variable is used in some cases but not others
  - *Extract Class,*
  - *Introduce Null Object*
- Message Chains: object asks another object, which asks another object, …
  - *Hide Delegate,*
  - *Extract Method,*
  - *Move Method*
- Middle Man: half of an object’s methods call other objects
  - *Remove Middle Man,*
  - *Inline Method,*
  - *Replace Delegation with Inheritance*

Watch out for comments!

- Comments are often used as deodorant on this smelly code
- Refactoring eliminates the smelly code so that the comments are superfluous
- Replace each commented block of code with a new method using *Extract Method* and *Introduce Assertion*

Refactoring Patterns

Some types of refactoring

- refactoring to fit design **patterns**
- **renaming** (methods, variables)
- **extracting** code into a method or module
- **splitting** one method into several to improve cohesion and readability
- changing method **signatures**
- performance **optimization**
- **moving** statements that semantically belong together near each other
- naming (extracting) "magic" **constants**
- **exchanging idioms** that are risky with safer alternatives
- **clarifying** a statement that has evolved over time or is unclear

  - See also http://www.refactoring.org/catalog/

Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

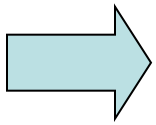
Refactoring patterns

- Composing Methods
  - **Extract Method**
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Extract Method

- Sometimes we have methods that do too much. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere. The Extract Method refactoring is one of the most useful for reducing the amount of duplication in code.

```
public class Customer
{
  void int foo()
  {
     …
     // Compute score
    score = a*b+c;
    score *= xfactor;
  }
}
```



```
public class Customer
{
  void int foo()
  {
     …
    score = ComputeScore(a,b,c,xfactor);
  }

  int ComputeScore(int a, int b, int c, int x)
  {
     return (a*b+c)*x;
  }
}
```

Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - **Replace Temp with Query**
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

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 and allows for other refactorings.

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

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

Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - **Move Method; Move Field**
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Move Method - Before

- If a method on one class uses (or is used by) another class more than the class on which its defined, move it to the other class

```
public class Student
{
  public boolean isTaking(Course course)
  {
     return (course.getStudents().contains(this));
   }
}

public class Course
{
  private List students;
  public List getStudents()
  {
    return students;
  }
}
```

Move Method - Refactored

- The student class now no longer needs to know about the Course interface, and the isTaking() method is closer to the data on which it relies - making the design of Course more cohesive and the overall design more loosely coupled

```
public class Student
{
}

public class Course
{
  private List students;
  public boolean isTaking(Student student)
  {
     return students.contains(student);
  }
}
```

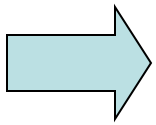
Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - **Extract Class**
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Extract Class

- Break one class into two, e.g., Having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle. We can refactor this into two separate classes, each with the appropriate responsibility.

```
public class Customer
{
 private String name;
 private String workPhoneAreaCode;
 private String workPhoneNumber;
}
```



```
public class Customer
{
private String name;
private Phone workPhone;
}

public class Phone
{
private String areaCode;
private String number;
}
```

Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - **Replace Nested Conditional with Guard Clauses**
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Replace Nested Conditional with Guard Clauses

- 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();
};
```

Refactoring patterns

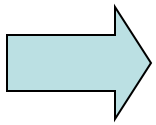
- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - **Introduce Null Object**
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Introduce Null Object

- If relying on null for default behavior, use inheritance instead

```
public class User
{
  Plan getPlan()
  {
      return plan;
  }
}
```

```
if (user == null)
    plan = Plan.basic();
else
   plan = user.getPlan();
```



```
public class User
{
  Plan getPlan()
  {
    return plan;
  }
}

public class NullUser extends User
{
  Plan getPlan()
  {
    return Plan.basic();
  }
}
```

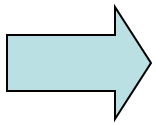
Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - **Rename Method**
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Rename Variable or Method

- Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose then change the name of the method or variable.

```
public class Customer
{
   public double getinvcdtlmt();
}
```



```
public class Customer
{
   public double getInvoiceCreditLimit();
}
```

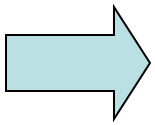
Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - **Replace Parameter with Explicit Methods**
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Replace Parameter with Explicit Method(s)

- 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();
}
```



```
void setHeight(int arg)
{
    height = arg;
}

void setWidth (int arg)
{
    width = arg;
}
```

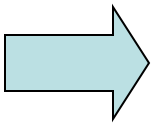
Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - **Replace Error Code with Exception**
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Replace Error Code with Exception

- A method returns a special code to indicate an error is better accomplished with an Exception.

```
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;
}
```

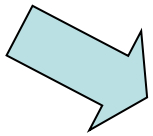
Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - **Replace Exception with Test**
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Replace Exception with Test

- Conversely, if you are catching an exception that could be handled by an if-statement, use that instead.

```
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];
 }
```

Refactoring patterns

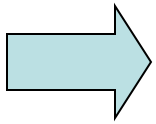
- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - **Extract Subclass; Extract Superclass; Interface**
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Extract Super Class

- When you find two or more classes that share common features, consider abstracting those shared features into a super-class. Again, this makes it easier to bind clients to an abstraction, and removes duplicate code from the original classes.

```
public class Employee
{
  private String name;
  private String jobTitle;
}

public class Student
{
  private String name;
  private Course course;
}
```



```
public abstract class Person
{
  protected String name;
}

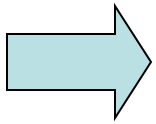
public class Employee extends Person
{
  private String jobTitle;
}

public class Student extends Person
{
  private Course course;
}
```

Extract Subclass

- When a class has features (attributes and methods) that would only be useful in specialized instances, we can create a specialization of that class and give it those features. This makes the original class less specialized (i.e., more abstract), and good design is about binding to abstractions wherever possible.

```
public class Person
{
 private String name;
 private String jobTitle;
}
```



```
public class Person
{
 protected String name;
}

public class Employee extends Person
{
 private String jobTitle;
}
```

Extract Interface

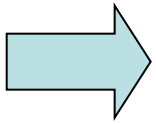
- Extract an interface from a class. Some clients may need to know a Customer's name, while others may only need to know that certain objects can be serialized to XML. Having toXml() as part of the Customer interface breaks the Interface Segregation design principle which tells us that it's better to have more specialized interfaces than to have one multi-purpose interface.

```
public class Customer
{
private String name;

public String getName(){ return name; }

public void setName(String string)
{ name = string; }

public String toXML()
{ return "<Customer><Name>" +
 name + "</Name></Customer>";
 }
}
```



```
public class Customer implements SerXML
{
private String name;

public String getName(){ return name; }

public void setName(String string)
{ name = string; }

public String toXML()
{ return "<Customer><Name>" +
 name + "</Name></Customer>";
 }
}
```

```
public interface SerXml {
   public abstract String toXML();
}
```

Refactoring patterns

- Composing Methods
  - Extract Method
  - Inline Method; Inline Temp
  - Replace Temp with Query
  - Introduce Explaining Variable
  - Split Temporary Variable
  - Remove Assignments to Parameters
  - Replace Method with Method Object
  - Substitute Algorithm
- Moving Features Between Objects
  - Move Method; Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middleman
  - Introduce Foreign Method
  - Introduce Local Extension
- Organizing Data
  - Self-encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference; Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to to Unidirectional
- Simplifying Conditional Expressions
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional with Polymorphism
  - Introduce Null Object
  - Introduce Assertion
- Making Method Calls Simpler
  - Rename Method
  - Add/Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter with Explicit Methods
  - Preserve Whole Object
  - Replace Parameter with Method
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor with Factory Method
  - Encapsulate Downcast
  - Replace Error Code with Exception
  - Replace Exception with Test
- Dealing with Generalization
  - Pull Up Field; Method; Constructor Body
  - Push Down Method; Push Down Field
  - Extract Subclass; Extract Superclass; Interface
  - Collapse Hierarchy
  - **Form Template Method**
  - Replace Inheritance with Delegation (or vice versa)
- Big Refactorings
  - Nature of the Game
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Form Template Method - Before

- When you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class

```
public abstract class Party { }
```

```
public class Person extends Party
{
 private String firstName;
 private String lastName;
 private Date dob;
 private String nationality;
 public void printNameAndDetails()
 {
   System.out.println("Name: " + firstName + " " + lastName);
   System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
  }
}
```

```
public class Company extends Party
{
  private String name;
  private String companyType;
  private Date incorporated;
  public void PrintNameAndDetails()
  {
   System.out.println("Name: " + name + " " + companyType);
   System.out.println("Incorporated: " + incorporated.toString());
  }
}
```

Form Template Method - Refactored

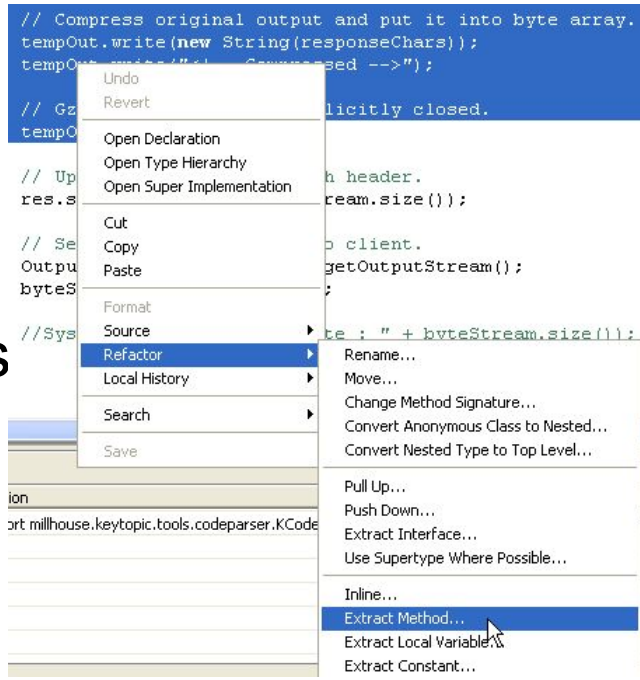
```
public abstract class Party
{
 public void PrintNameAndDetails()
 {
   printName();
   printDetails();
 }
 public abstract void printName();
 public abstract void printDetails();
}

public class Person extends Party
{
  private String firstName;
  private String lastName;
  private Date dob;
  private String nationality;
  public void printDetails()
  {
    System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
  }
  public void printName()
  {
    System.out.println("Name: " + firstName + " " + lastName);
  }
}
```

```
public class Company extends Party
{
  private String name;
  private String companyType;
  private Date incorporated;
  public void printDetails()
  {
    System.out.println("Incorporated: " + incorporated.toString());
  }
  public void printName()
  {
    System.out.println("Name: " + name + " " + companyType);
  }
}
```

IDE support for refactoring

- Eclipse and Visual Studio support:
  - variable / method / class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion



Refactoring plan

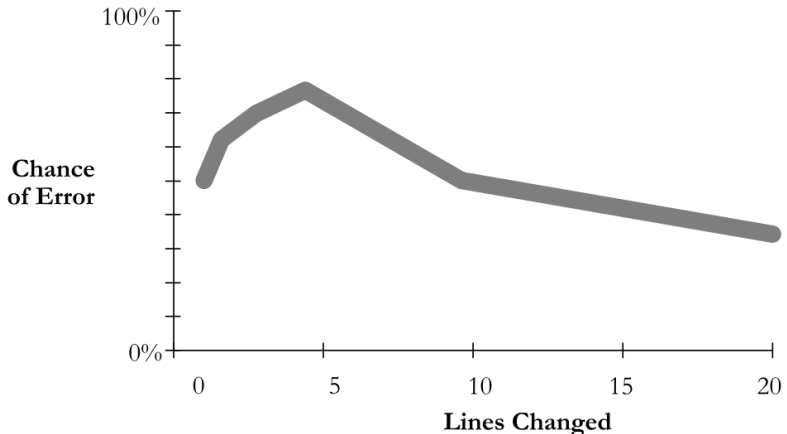
- Save / **backup** / check-in the code before you mess with it.
  - If you use a well-managed version control repo, this is done.
- Write **unit tests** that verify the code's external correctness.
  - They should pass on the current poorly designed code.
  - Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).
- Analyze the code to decide the **risk** and benefit of refactoring.
  - If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.
- **Refactor** the code.
  - Some unit tests may break. Fix the bugs.
  - Perform functional and/or integration testing. Fix any issues.
- **Code review** the changes.
- **Check-in** your refactored code.
  - Keep each refactoring **small**; refactor one issue / unit at a time.
    - helps isolate new bugs and regressions
  - Your check-in should contain *only* your refactored code.
    - NOT other changes such as adding features, fixing unrelated bugs.
    - Do those in a separate check-in.
      - (Resist temptation to fix small bugs or other tweaks; this is dangerous.)

"I don't have time!"

- Refactoring incurs an **up-front cost**.
  - many developers don't want to do it
  - management don't like it; they lose time and gain "nothing“ (no new features)
- But...
  - well-written code is more conducive to **rapid development**
    - Some estimates put ROI at 500% or more for well-done code
  - refactoring is good for **programmer morale**
    - developers prefer working in a "clean house"

Dangers of refactoring

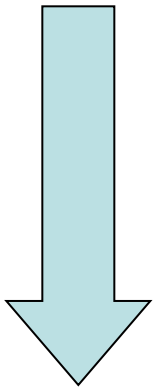
- code that used to be ...
  - well commented, now (maybe) isn't
  - fully tested, now (maybe) isn't
  - fully code reviewed, now (maybe) isn't
- easy to insert a bug into previously working code (regression!)
  - a small initial change can have a large chance of error



Recall: switch statements (before)

```
class Animal {
   final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
   int myKind;  // set in constructor
   ...
   String getSkin() {
      switch (myKind) {
         case MAMMAL: return "hair";
         case BIRD: return "feathers";
         case REPTILE: return "scales";
         default: return “skin";
      }
   }
}
```

Switch Statements



Recall:
switch statements (improved)

```
class Animal {
    String getSkin() { return “skin"; }
}
class Mammal extends Animal {
    String getSkin() { return "hair"; }
}
class Bird extends Animal {
    String getSkin() { return "feathers"; }
}
class Reptile extends Animal {
    String getSkin() { return "scales"; }
}
```

JUnit tests

- As we refactor, we need to run JUnit tests to ensure that we haven't introduced errors
- ```
  public void testGetSkin() {
      assertEquals("hair", myMammal.getSkin());
      assertEquals("feathers", myBird.getSkin());
      assertEquals("scales", myReptile.getSkin());
      assertEquals("integument", myAnimal.getSkin());
  }
  ```
- This should work equally well with either implementation
- The setUp() method of the test fixture may need to be modified
  ```

Regressions

What if refactoring introduces new bugs in previously working functionality ("regressions")? How can this be avoided?

- Code being refactored should have good **unit test** coverage, and other tests (system, integration) over it, *before* the refactor.
  - If such code is not tested, **add tests** first *before* refactoring.
  - If the refactor makes a unit test not compile, **port it**.
  - If the method being tested goes away, the underlying functionality of that method should still be somewhere. So move the unit test to cover that new place in the code.

Company/team culture

- **Organizational barriers** to refactoring:
  - Many small companies and startups don't do it.
    - "We're too small to need it!" ... or,
    - "We can't afford it!"
  - Many larger companies don't adequately reward it.
    - Not as flashy as adding features/apps;
    - ignored at promotion time
- Reality:
  - Refactoring is an investment in quality of the company's product and code base, often their prime assets.
  - Many web startups are using the most cutting-edge technologies, which evolve rapidly. So should the code.
  - If a team member leaves or joins (common in startups), ...
  - Some companies (e.g. **Google**) actively reward refactoring.

Refactoring and teams

- Amount of overhead/communication needed depends on size of refactor.
  - *small refactor:* Just do it, check it in, get it code reviewed.
  - *medium refactor:* Possibly loop in tech lead or another dev.
  - *large refactor:* Meet with team, flush out ideas, do a design doc or design review, get approval before beginning.
- Avoids possible bad scenarios:
  - Two devs refactor same code simultaneously.
  - Refactor breaks another dev's new feature they are adding.
  - Refactor actually is not a very good design; doesn't help.
  - Refactor ignores future use cases, needs of code/app.
  - Tons of merge conflicts and pain for other devs.

Phased refactoring

- Sometimes a refactor is too big to do all at once.
  - Example: An entire large subsystem needs redesigning. We don't think we have time to redo all of it at once.
- **Phased refactoring**: Adding a **layer of abstraction** on top of legacy code.
  - New well-made System 2 on top of poorly made old System 1.
  - System 1 remains; Direct access to it is *deprecated*.
  - For now, System 2 still forwards some calls down to System 1 to achieve feature parity.
  - Over time, calls to System 1 code are replaced by new System 2 code providing the same functionality with a better design.

Refactoring at Google

- "At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do *small* refactoring tasks early and often, as soon as there is a sign of a problem."
- "Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts."
- "Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded."
- "Common reasons not to do it are incorrect:
  - a) *'Don't have time; features more important'* -- You will pay more cost, time in adding features (because it's painful in current design), fixing bugs (because bad code is easy to add bugs into), ramping up others on code base (because bad code is hard to read), and adding tests (because bad code is hard to test), etc.
  - b) *'We might break something'* -- Sign of a poor design from the beginning, where you didn't have good tests. For same reasons as above, you should fix your testing situation and code.
  - c) *'I want to get promoted and companies don't recognize refactoring work'* -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality."
- "An important line of defense against introducing new bugs in a refactor is having solid unit tests (*before* the refactor)."

-- Victoria Kirst,
Software Engineer, Google