Software Design and Architecture

Introduction to Refactoring

What is Refactoring

Refactoring is the process of changing a software system such that

- the external behavior of the system does not change
  - e.g. functional requirements are maintained
- but the internal structure of the system is improved

This is sometimes called

- "**Improving the design after it has been written**"

The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time and, as a program's requirements change, the design may need to change

- refactoring provides techniques for evolving the design in small incremental steps

(Very) Simple Example

Consolidate Duplicate Conditional Fragments; This

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

becomes this

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

(Another) Simple Example

Replace Magic Number with Symbolic Constant

```
double potentialEnergy(double mass, double height) {
return mass * 9.81 * height;
}
```

becomes this

```
double potentialEnergy(double mass, double height) {
return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Refactoring is thus Dangerous!

Manager's point-of-view

- If my programmers spend time "cleaning up the code" then that's less time implementing required functionality (and my schedule is slipping as it is!)

Refactoring needs to be **systematic, incremental, and safe**

Refactoring Benefits

Often code size is reduced after a refactoring

Confusing structures are transformed into simpler structures

- which are easier to maintain and understand

A “cookbook” can be useful

- **Refactoring: Improving the Design of Existing Code**
  - by Martin Fowler (and Kent Beck, John Brant, William Opdyke, and Don Roberts)
- Similar to the Gang of Four’s Design Patterns
  - Provides “refactoring patterns”

Principles in Refactoring

Fowler's definition

Refactoring (noun)

- a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

Refactoring (verb)

- to restructure software by applying a series of refactorings without changing its observable behavior

Principles in Refactoring

The purpose of **refactoring** is

- to make software easier to understand and modify

contrast this with **performance optimization**

- again functionality is not changed, only internal structure;
- however performance optimizations often involve making code harder to understand (but faster!)

Useful Reference for Code Performance Optimization

Code Complete: A Practical Handbook of Software Construction, Second Edition 2nd Edition

Principles in Refactoring

When you systematically apply refactoring, you wear two hats

adding function

- **functionality is added** to the system **without** spending any time **cleaning the code**

refactoring

- **no functionality is added**, but the code is **cleaned up**, made easier to understand and modify, and sometimes is reduced in size

How do you make refactoring safe?

First, use refactoring “patterns”

- Fowler’s book assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns

How do you make refactoring safe?

Second, test constantly!

This ties into the **agile design paradigm**

- you write tests before you write code
- after you refactor, you run the tests and check that they all pass
  - if a test fails, the refactoring broke something but you know about it right away and can fix the problem before you move on

Why should you refactor?

- **Refactoring improves the design of software**
  - without refactoring, a design will "decay" as people make changes to a software system
- **Refactoring makes software easier to understand**
  - because structure is improved, duplicated code is eliminated, etc.

Why should you refactor?

- **Refactoring helps you find bugs**
  - Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs
- **Refactoring helps you program faster**
  - because a good design enables progress

When should you refactor?

- **Refactor when you add functionality**
  - do it before you add the new function to make it easier to add the function
  - or do it after to clean up the code after the function is added
- **Refactor when you need to fix a bug**
- **Refactor as you do a code review**

Problems with Refactoring

Business applications are often tightly coupled to underlying databases

- code is easy to change; databases are not

Changing Interfaces (!!)

- Some refactorings require that interfaces be changed
  - if you own all the calling code, no problem
  - if not, the interface is “published” and can’t change

Major design changes cannot be accomplished via refactoring

  - This is why agile design says that software devs. need courage!

Refactoring: Where to Start?

How do you identify code that needs to be refactored?

- **Fowler uses an olfactory analogy (attributed to Kent Beck)**
- **Look for "Bad Smells" in Code**
  - A very valuable chapter in Fowler's book
  - It presents examples of "bad smells" and then suggests refactoring techniques to apply

Bad Smells in Code

- **Duplicated Code**
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- **Long Method**
  - long methods are more difficult to understand
    - performance concerns with respect to lots of short methods are largely obsolete

Bad Smells in Code

- **Large Class**
  - Large classes try to do too much, which reduces cohesion
- **Long Parameter List**
  - hard to understand, can become inconsistent if the same parameter chain is being passed from method to method
- **Divergent Change**
  - **symptom:** one type of change requires changing one subset of methods; another type of change requires changing another subset
  - **Related to cohesion**

Bad Smells in Code

- **Shotgun Surgery**
  - a change requires lots of little changes in a lot of different classes
- **Feature Envy**
  - A method requires lots of information from some other class
- **Data Clumps**
  - attributes that clump together (are used together) but are not part of the same class

Bad Smells in Code

- **Primitive Obsession**
  - characterized by a reluctance to use classes instead of primitive data types
- **Switch Statements**
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
- **Parallel Inheritance Hierarchies**
  - Similar to Shotgun Surgery; each time we add a subclass to one hierarchy, we need to do it for all related hierarchies
    - **Note:** some design patterns encourage the creation of parallel inheritance hierarchies (so they are not always bad!)

Bad Smells in Code

- **Lazy Class**
  - A class that no longer “pays its way”
  - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- **Speculative Generality**
  - “Oh we think we need the ability to do this kind of thing someday”
- **Temporary Field**
  - An attribute of an object is only set/used in certain circumstances;
    - but an object should need all of its attributes

Bad Smells in Code

- **Message Chains**
  - a client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation
- **Middle Man**
  - If a class is delegating more than half of its responsibilities to another class, do you really need it? Involves trade-offs, some design patterns encourage this (e.g. Decorator)
- **Inappropriate Intimacy**
  - Pairs of classes that know too much about each other's implementation details (loss of encapsulation; change one class, the other has to change)

Bad Smells in Code

- **Data Class (information holder)**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
  - Subclass may not pass the “IS-A” test
- **Comments (!)**
  - Comments are sometimes used to hide bad code
    - “…comments often are used as a deodorant” (!)

The Catalog

The refactoring book has 72 refactoring patterns!

Some of common ones:

- Extract Method
- Replace Temp with Query
- Move Method
- Replace Conditional with Polymorphism
- Introduce Null Object
- Separate Query for Modifier
- Introduce Parameter Object
- Encapsulate Collection
- Replace Nested Conditional with Guard Clauses

Extract Method

- You have a code fragment that can be grouped together
- Turn the fragment into a method whose name explains the purpose of the fragment

Example

```
void printOwing(double amount) {
    printBanner()
    //print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + amount);
}

==========================================

void printOwing(double amount) {
    printBanner()
    printDetails(amount)
}

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

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

Example

```
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;
}
```

Move Method

- A method is using more features (attributes and operations) 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

Move Method

```
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;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

A class to manage a bank account. There are currently two types of accounts: standard and premium.

Currently we have only two types of accounts, **standard** and **premium**, but it is anticipated that we will be adding new account types and that each type will have a different rule for calculating an overdraft charge.

As such, we'd like to **move the method** overdraftCharge() **to** the AccountType class.

Move Method

When moving a method to a new class, we examine its code to see if makes use of internal attributes of its original class

- In this case, overdraftCharge() makes use of _daysOverdrawn

All such attributes become parameters to the method in its new home. (If the method already had parameters, the new parameters get tacked on to the end of its existing parameter list.)

- In this case, _daysOverdrawn will stay in the Account class and be passed as a parameter to AccountType.overdraftCharge().

Move Method

Note, also, that since we are moving this method to the AccountType class, all calls to its methods that previously required a variable reference, can now be made directly

- Thus, _type.isPremium() becomes simply isPremium() in the method's new home

Move Method

```
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;
        }
    }
    ...
}
```

Here is the method in its new home. It has a daysOverdrawn parameter, which is used instead of _daysOverdrawn, throughout the method. _type.isPremium() is now just isPremium(), as advertised.

Move Method

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

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

    private AccountType _type;
    private int _daysOverdrawn;
}
```

Back in the Account class, we update overdraftCharge() to delegate to the overdraftCharge() method in the AccountType class. Or, we could…

Move Method

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

    private AccountType _type;
    private int _daysOverdrawn;
}
```

… get rid of the overdraftCharge() method in Account entirely.

In that case, we move the call to AccountType.overdraftCharge() to bankCharge()

Replace Conditional with Polymorphism

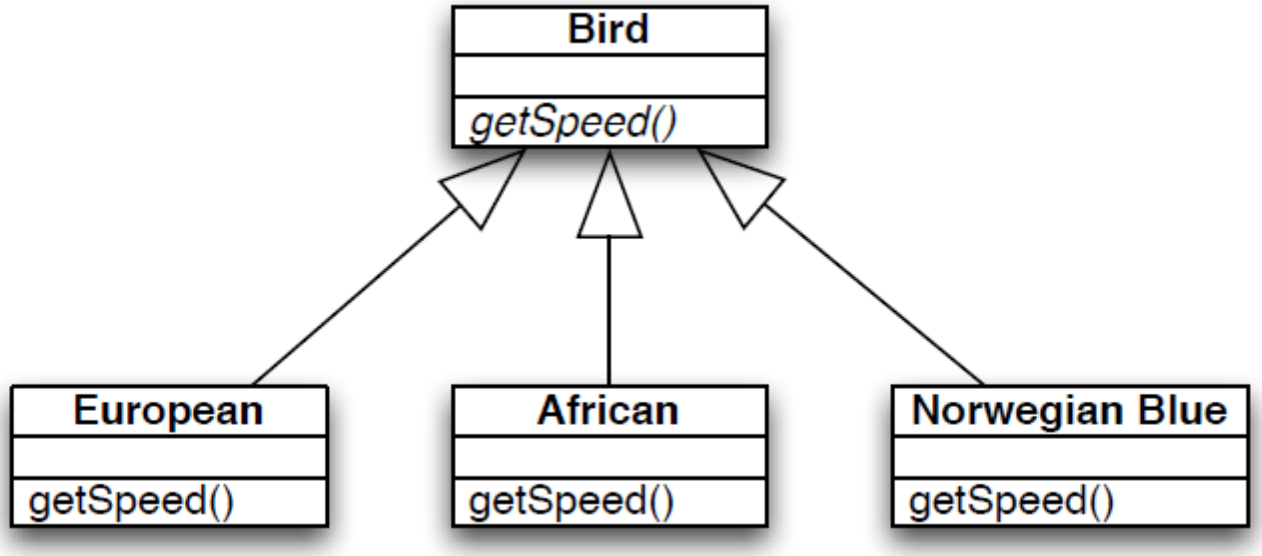
You have a conditional that chooses different behavior depending on the type of an object

- Move each “leg” of the conditional to an overriding method in a subclass. Make the original method abstract

Replace Conditional with Polymorphism

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("Unknown Type of Bird")
}
```

Replace Conditional with Polymorphism



With this configuration, you can now write code that looks like this:

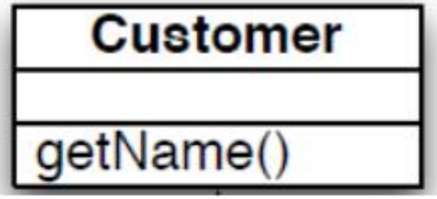
```
void printSpeed(Bird[] birds) {
    for (int i=0; i < birds.length; i++) {
        System.out.println("" + birds[i].getSpeed());
    }
}
```

and everything will work correctly via polymorphism and will be easy to extend: just add a new subclass to support a new type of bird

Introduce Null Object

- Repeated checks for a null value (see below)
- Rather than returning a null value from findCustomer() return an instance of a “null customer” object

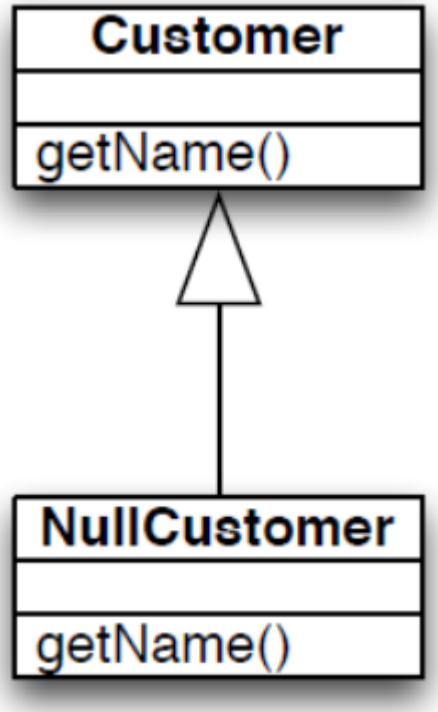
```
...
Customer c = findCustomer(...);
...
if (customer == null) {
    name = “occupant”
} else {
    name = customer.getName()
}
if (customer == null) {
...
```



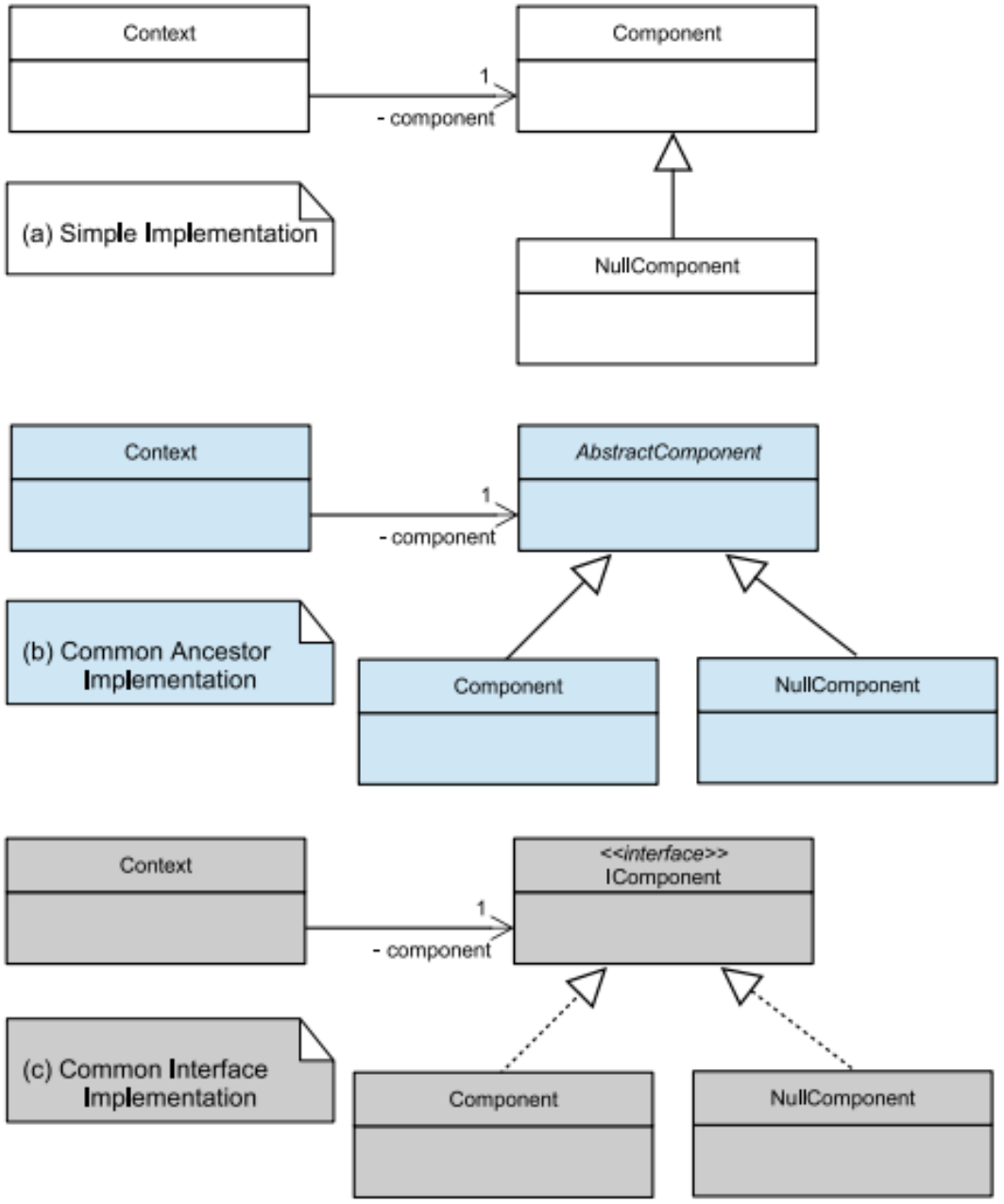
Introduce Null Object

The conditional goes away entirely!!

```
public class NullCustomer {
  public String getName() { return "occupant";}
}
============================
Customer c = findCustomer(...);
name = c.getName();
```



Null Object Design Pattern



Separate Query for Modifier

Sometimes you will encounter code that does something like this

- **get**TotalOutstandingAnd**Set**ReadyForSummaries()

It is a query method but it is also changing the state of the object being called

- This change is known as a “side effect” because it’s not the primary purpose of the method

Separate Query for Modifier

It is generally accepted practice that queries should not have side effects so this refactoring says to split methods like this into:

- **get**TotalOutstanding()
- **set**ReadyForSummaries()

Try as best as possible to avoid side effects in query methods

Introduce Parameter Object

You have a group of parameters that go naturally together

- Stick them in an object and pass the object

Imagine methods like

- amountInvoicedIn(start: Date; end: Date);
- amountOverdueIn(start: Date; end: Date);

This refactoring says replace them with something like

- amountInvoicedIn(**dateRange: DateRange**)
- The new class starts out as a data holder but will likely attract methods to it

Encapsulate Collection

A method returns a collection

- Make it return a read-only version of the collection and provide add/remove methods

Student class with

- getCourses(): Map;
- setCourses(courses: Map);

Change to

- getCourses(): ReadOnlyList
- addCourse(c : Course)
- removeCourse(c : Course)

Changing the externally visible collection, too, is a good idea to protect clients from depending on the internals of the Student class

Replace Nested Conditional with Guard Clauses

This refactoring relates to the purpose of conditional code

- One type of conditional checks for a variation in “normal” behavior
  - The system will either do A or B, both are considered “normal” behavior
- The other type of conditional checks for unusual circumstances that require special behavior; if all of these checks fail then the system proceeds with “normal behavior”

We want to apply this refactoring when we discover the latter type of conditional

Example

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

Note: This type of code may be the result of a novice programmer or due to a programming constraint imposed by some companies that a method can only have a single return.

Often this constraint causes more confusion than its worth

Example

```
double getAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalAmount();
}
```

With this refactoring, all of the code trying to identify special conditions are turned into one-line statements that determine whether the condition applies and if so handles it

That's why these statements are called "guard clauses"

Even though this method has four returns, its easier to understand than the method before the refactoring

Common Situation: Need to reverse conditionals

Sometimes complex compound conditionals can be converted into guard clauses by reversing the condition that is being tested

By reversing the conditionals and "failing early" you simplify the code

Example

```
double getAdjustedCapital() {
  double result = 0.0;
  if (_capital > 0.0) {
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
  }
  return result;
}
==============================
double getAdjustedCapital() {
  double result = 0.0;
  if (_capital <= 0.0) return result;
  if (_intRate > 0.0 && _duration > 0.0) {
    result = (_income / _duration) * ADJ_FACTOR;
  }
  return result;
}
```

Example

```
double getAdjustedCapital() {
  double result = 0.0;
  if (_capital <= 0.0) return result;
  if (!(_intRate > 0.0 && _duration > 0.0)) return result;
  result = (_income / _duration) * ADJ_FACTOR;
  return result;
}
==============================
double getAdjustedCapital() {
  double result = 0.0;
  if (_capital <= 0.0) return result;
  if (_intRate <= 0.0 || _duration <== 0.0) return result;
  result = (_income / _duration) * ADJ_FACTOR;
  return result;
}
```

Example

```
static double NO_ADJUSTMENT = 0.0;
double getAdjustedCapital() {
  if (_capital <= 0.0) return NO_ADJUSTMENT;
  if (_intRate <= 0.0 || _duration <= 0.0)) return NO_ADJUSTMENT;
  return (_income / _duration) * ADJ_FACTOR;
}
```

In this final step
we can eliminate the temporary variable altogether, replace 0.0 with a constant that provides semantic meaning, and just return the adjusted capital without storing it in a temp first.

The resulting method is simpler, shorter, easier to understand