Refactoring Part 2

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand how refactorings are applied
- Apply these refactorings in your daily practice

Inspiration

"A little retrospection shows that although many fine, useful software systems have been designed by committees and built as part of multipart projects, those software systems that have excited passionate fans are those that are the products of one or a few designing minds, great designers." – Fred Brooks

The Catalog of Refactorings

- Fowler's book and Website (http://www.refactoring.com/catalog/) has 72+ refactoring patterns
  - I'm only going to cover a few of the more common ones, including
    - 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, next slide

Extract Method

This

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

Becomes This

```
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 an expression
  - The new method can then be used in other methods
- Example, next slide

Idaho State University
Computer Science

Replace Temp with Query

This

```
double basePrice = quantity * itemPrice;

if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

Becomes This

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


ROAR

Move Method (I)

- 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 (II)

```
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

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 (III)

- When moving a method to a new class, we examine its code to see if it 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()`.
- 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 (IV)

```
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 is 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 (V)

```
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 (VI)

```
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()`

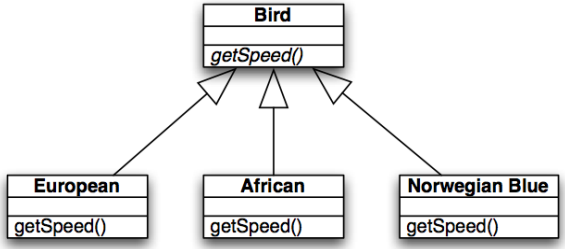
Replace Conditional with Polymorphism

- 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 getBseSpeed() - getLoadFactor() * numCoconuts;
    case NORWEGIAN_BLUE:
      reutrn (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:

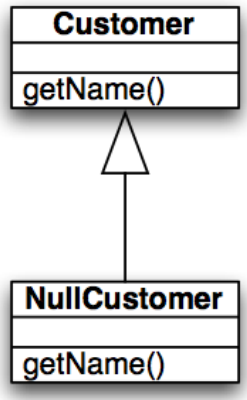
```
void printSpeed(Bird[] birds) {
  for (int i = 0; i < birds.length; i++) {
    System.out.println("" + birds[i].getSpeed());
  }
}
```

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 (I)

- 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 (II)

```
public class NullCustomer {
  public String getName() { return "occupant"; }
}

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

Customer c = findCustomer(...);
name = c.getName();
```

- The conditional goes away entirely!

Separate Query for Modifier

- Sometimes you will encounter code that does something like this
  - `getTotalOutstandingAndSetReadyForSummaries()`
- 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
- It is generally accepted practice that queries should not have side effects so this refactoring says to split methods like this into:
  - `getTotalOutstanding()`
  - `setReadyForSummaries()`
- Try as best as possible to avoid any 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(Data start, Date end)`
  - `amountOverdueIn(Date start, Date end)`
- 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
  - `Map getCourses()`
  - `void setCourses(Map courses)`
- Chage to
  - `ReadOnlyList getCourses()`
  - `addCourse(Course c)`
  - `removeCourse(Course c)`

Replace Nested Conditional

- This refactoring relates to the purpose of conditional code
  - Only type of conditional checks for a variation in “normal” behavior
    - The system will do either 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 encounter the latter type of conditional
- This refactoring is described in Fowler’s book as:
  - “A method has conditional behavior that does not make clear the normal path of execution; Use guard clauses for all special cases”

Example (I)

```
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 (II)

```
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, it is easier to understand than the method before the refactoring

Wrapping Up

- Refactoring is a useful technique for making non-functional changes to a software system that result in
  - better code structures
  - less code
    - Many refactorings are triggered via the discovery of duplicated code
    - The refactorings then show you how to eliminate duplication
- Bad Smells
  - Useful analogy for discovering places in a system “ripe” for refactoring

Are there any questions?