Refactoring, cont.

Announcements

- HW8 due December 1st
  - First, refactor
  - Then implement NOT Expression
  - Last, implement Visitors: Evaluate, PrintInorder
- Check your grades
  - Quiz 1-7, Exam 1-2, HW0-5 in Submitty
  - Pay attention if you have excused late days
- Quiz 8 and Quiz 9 today

Outline of Today’s Class

- Refactorings
  - We’ll review the State, Strategy and Template Method design patterns
  - Extract method, Move method
  - Replace temp with query
  - Replace Type Code with State/Strategy
  - Replace Conditional with Polymorphism
  - Form Template Method
  - Replace Magic Number with Symbolic Constant
- Refactoring, Conclusion

Refactoring

- Small-step, semantics-preserving transformation
- Continuous testing
- Goal: achieve code that is short, tight, clear and without duplication. Eliminate code smells
- Refactorings
  - Extract method
  - Move method
  - Replace temp with query… More

Before…

- Code smells: all code in long method statement(), unnecessary coupling between Customer and Rental and Customer and Movie

After…

- Shortened statement() with Extract Method, eliminated unnecessary coupling between Customer and Rental, and Customer and Movie with Move Method, improved readability with Replace Temp with Query
Customer.statement, Refactored

```java
public String statement() {
    String result = "Rental Record for " + getName() + "\n";
    for (Rental each : _rentals) {
        result += "...+String.valueOf(each.getCharge())+...\n";
    }
    result += "...+getTotalCharge() + ... +
    getTotalFrequentRenterPoints()\n";
    return result;
}
```

Added a Method

```java
public String htmlStatement() {
    String result = "<H1>Rental Record for <EM>
    " + getName() + "</EM></H1><P>
    ";
    for (Rental each : _rentals) {
        result += "...+each.getCharge()+...\n"; // add HTML
    }
    result += "... +getTotalCharge() +
    +getFrequentRenterPoints() + HTML
    
    return result;
}
```

First Refactoring: Move Method into “Information Expert”

- Problem: A switch statement on own data is bad. A switch statement on someone else's data is worse
- Move `getCharge` and `getFrequentRenterPoints` from class `Rental` to class `Movie`:
  ```java
class Rental { // replace with delegation
    double getCharge() {
        return _movie.getCharge(_daysRented);
    }
    ...
}
```

Key point: refactoring is intertwined with addition of new methods and functionality. What's the problem here?

Still refactoring... Back to `getCharge`

```java
double getCharge() { // now in Rental
double result = 0;
switch (getMovie().getPriceCode()) { // Switch over Movie's data
case Movie.REGULAR:
    result +=2;
    if (getDaysRented()>2)
        result += (getDaysRented()-2)*1.5;
    break;
case Movie.NEW_RELEASE:
    result += getDaysRented()*3;
    break;
case Movie.CHILDRENS:
    result += 1.5;
    if (getDaysRented()>3)
        result += (getDaysRented()-3)*1.5;
    break;
} return result;
```

Fall 17 CSCI 2600, A Milanova

What's wrong here?

Second Step: Get Rid of Switch

- Problem: a switch statement is a bad idea, it is difficult to maintain and error prone
- Solution: replace switch with subtype polymorphism!
  - Abstract class `Price` with concrete subclasses `Regular`, `Childrens`, `NewRelease`
  - Each `Price` subclass defines its own
    - getPriceCode()
    - getCharge()
Aside: the **Strategy** Design Pattern

- Question: Can we have an algorithm vary independently from the object that uses it?
  - Example: Movie pricing…
    - Class Movie represents a movie
    - There are several pricing algorithms/strategies
    - Placing pricing algorithms/strategies in Movie makes Movie complex and inflexible: the switch statement

```
class Movie {
    // Pricing algorithms/strategies
    // Switch statement
}
```

Aside: the **State** Design Pattern

- Question: How can an object alter its behavior when its internal state changes?
  - Example: A TCPConnection class representing a network connection
    - TCPConnection can be in one of three states: Established, Listening, or Closed
    - When a TCPConnection object receives requests (e.g., open, close) from the client, it responds differently depending on its current state

```
class TCPConnection {
    private TCPState state;
    // Other methods...
    public void open() {
        state = TCPEstablished.getInstance();
    }
    // Other methods...
}
```

### The Strategy Pattern

- Price: Can we have an algorithm vary independently from the object that uses it?
- Example: Movie pricing...
- There are several pricing algorithms/strategies
- Have to add/change algorithms/strategies easily
- Placing pricing algorithms/strategies in Movie makes Movie complex and inflexible: the switch statement

```
class Movie {
    private Price _price;
    // Other methods...
    public void getPriceCode() {
        return _price.getPriceCode();
    }
}
```

```
interface Price {
    int getPriceCode();
    int getCharge(int); // Many cases...
}
```

#### Step 1: Replace Type Code with State/Strategy refactoring
- Creates Price State/Strategy hierarchy
- Move code from Movie to Price, Movie delegates: 
  - _price.getPriceCode(...) 
- Step 2: Replace Conditional with Polymorphism
- Move case branches from Price to subclasses
- Calls to _price become virtual: _price.getPriceCode(...)

#### The State Pattern

- TCPConnection can be in one of three states: Established, Listening, or Closed
- When a TCPConnection object receives requests (e.g., open, close) from the client, it responds differently depending on its current state

```
class TCPConnection {
    private TCPState state;
    // Other methods...
    public void open() {
        state = TCPEstablished.getInstance();
    }
    // Other methods...
}
```

```
class TCPEstablished extends TCPState {
    // Method overrides...
}
```

```
class TCPListening extends TCPState {
    // Method overrides...
}
```

```
class TCPClosed extends TCPState {
    // Method overrides...
}
```

### Back to Getting Rid of Switch

- Need coupling from TCPState to TCPConnection: when state changes, field “state” must be properly updated
- State classes are often Singletons

```
class TCPConnection {
    private TCPState state;
    // Other methods...
    public void open() {
        state = TCPEstablished.getInstance();
    }
    // Other methods...
}
```

```
class TCPEstablished extends TCPState {
    public void open() {
        close();
        acknowledge();
    }
    // Other overrides...
}
```

```
class TCPListening extends TCPState {
    public void open() {
        close();
        acknowledge();
    }
    // Other overrides...
}
```

```
class TCPClosed extends TCPState {
    public void open() {
        close();
        acknowledge();
    }
    // Other overrides...
}
```
The **Replace Type Code with State/Strategy Refactoring**

- Add the new *Price* hierarchy
- In *Movie*: `int _priceCode` becomes *Price* *_price*
- Change *Movie’s* accessors to use `_price`
  - `int getPriceCode() { return _price.getPriceCode(); }`
  - `void setPriceCode(int arg) {
      switch (arg) {
        case REGULAR: _price = new Regular();
        … }
  }

**Move Method** `getCharge()` from *Movie* to *Price*

```java
double getCharge(int daysRented) { // now in Price…
  double result = 0;
  switch (getPriceCode()) { // Note this stays the same!
  case REGULAR:
    result += 2;
    if (daysRented > 2) result += (daysRented - 2) * 1.5;
    break;
  case NEW_RELEASE:
    result += daysRented * 3;
    break;
  case CHILDRENS:
    result += 1.5;
    if (daysRented > 3) result += (daysRented - 3) * 1.5;
    break;
  }
  return result;
}
```

The **Replace Conditional with Polymorphism Refactoring**

- *Regular* defines its `getCharge`
  - `double getCharge(int daysRented) {
      double result = 2;
      if (daysRented > 2) result += (daysRented - 2) * 1.5;
      return result;
    }
  `
- *Childrens* and *NewRelease* define their `getCharge(int)`
- `getCharge(int)` in *Price* becomes abstract

**So Far**

- Extract Method
- Move Method
- Replace Temp with Query
- Replace Type Code with State/Strategy and Replace Conditional with Polymorphism
- Last two refactorings go together, break transformation into small steps
- **Goal**: replace switch with subtype polymorphism

**Still Refactoring…**

```java
public String statement() { // in Customer
  String result = "Rental Record for " + getName() + "\n";
  for (Rental each : _rentals) {
    result += "...\n" + each.getCharge() + "\n";
  }
  result += "...\n" + getTotalCharge() + "\n";
  result += "...\n" + getTotalFrequentRenterPoints() + "\n";
  return result;
}
```

At some point, we created `htmlStatement()` which was the same, except that `result` had HTML symbols.
Still Refactoring...

```java
public String htmlStatement() { // in Customer
    String result = "<H1>Rental Record for <EM>" + getName() + "</EM><H1><P>
    for (Rental each : _rentals) {
        result += each.getCharge() + "\n"; // + HTML
    }
    result += ... + getTotalCharge() + ... + getFrequentRenterPoints(); // + HTML
    return result;
}
```

We created htmlStatement using cut-and-paste. Code smell?

There is a design pattern that helps deal with duplicate code!

Before we deal with duplicate code...

- **Introduce Strategy for printing statements**
  - abstract class Statement
  - class TextStatement extends Statement
  - class HtmlStatement extends Statement

- **Move Method**
  - Customer.statement() to TextStatement.value(Customer)
  - Customer.htmlStatement() to HtmlStatement.value(Customer)

- **Delegation in Customer**
  - String statement() { return (new TextStatement()).value(this); }
  - String htmlStatement() { return (new HtmlStatement()).value(this); }

Aside: The Template Method Design Pattern

- Problem: We have several methods that implement the same algorithm, but differ at some steps
  - E.g., TextStatement.value and HtmlStatement.value
- Solution: Define the skeleton of the algorithm in a superclass, defer differing steps to subclasses
- Example: TextStatement and HtmlStatement
  - Same algorithm for TextStatement.value and HtmlStatement.value:
    - First, record header substring: customer info
    - Iterate over rentals, record each rental substring
    - Finally, record footer substring: total charge
  - Recorded substrings differ from Text to Html

Aside: The Template Method Pattern

- value is the template method
- headerString, rentalString, footerString are hooks
- Hooks are abstract in Statement defer to subclasses

Question

- Where is the template method? Hooks?

The Form Template Method Refactoring

- **Before refactoring** TextStatement.value and HtmlStatement.value are very similar
  - The “duplicate code” smell
- **Refactor to form a template method**
  - Eliminates the “duplicate code” smell
The Form Template Method Refactoring

- Decompose the methods using **Extract Method** so that all extracted methods are either identical among the different subclasses, or completely different.
- Use **Pull Up Method** (another refactoring!) to pull the identical methods, from one subclass, into the superclass.
- For the different methods use **Rename Method**
  - Make sure that each one has the same name+signature
  - Compile and test after the signature changes
  - Declare them as abstract in the superclass
  - Remove all identical methods, compile and test after each removal.

Form Template Method Step 1: Extract Method

class TextStatement extends Statement { …
public String value(Customer c) {
    String result = "Rental Record for " + getName() + 
    "\n" + headerString(c);
    for (Rental each : _rentals) {
        result += eachRentalString(each);
    }
    result += footerString(c);
    return result;
}

For the different methods use **Pull Up Method**

Now, **Pull Up** value(Customer) from TextStatement into Statement:

abstract class Statement { …
public String value(Customer c) {
    String result = headerString(c);
    for (Rental each : _rentals) {
        result += eachRentalString(each);
    }
    result += footerString(c);
    return result;
}

For the different methods use **Rename Method**

- Compile and test after the signature changes
- Declare them as abstract in the superclass
- Remove all identical methods, compile and test after each removal.

Form Template Method Step 2: Pull Up Method

Now, Pull Up value(Customer) from TextStatement into Statement:

abstract class Statement { …
public String value(Customer c) {
    String result = headerString(c);
    for (Rental each : _rentals) {
        result += eachRentalString(each);
    }
    result += footerString(c);
    return result;
}

Step 3: Make the hooks abstract in Statement

abstract String headerString(Customer c);
abstract String eachRentalString(Rental each);
abstract String footerString(Customer c);

Step 4: Compile and test!

Step 5: Remove value(Customer) from HtmlStatement as well. Compile and test!

There is coupling from Statement to Customer, value needs rentals and name from Customer.

One last refactoring. Back to getCharge(int)

class Regular extends Price { …
double getCharge(int daysRented) { …
    if (daysRented > 2) {
        result += (daysRented - 2)*1.5;
    }
    return result;
}
Replace Magic Number with Symbolic Constant

```java
class Regular extends Price {
    final static double INTRO_DAYS = 2;
    final static double INTRO_RATE = 1;
    final static double REGULAR_RATE = 1.5;

    double getCharge(int daysRented) {
        double result = INTRO_DAYS * INTRO_RATE;
        if (daysRented > INTRO_DAYS)
            result += (daysRented - INTRO_DAYS) * REGULAR_RATE;
        return result;
    }
}
```

Recall Replace Temp with Query

```java
public String statement() {
    String result = "Rental Record for " + getName() + ", ";
    for (Rental each : _rentals) {
        result += "...+String.valueOf(each.getCharge())+"...";
    }
    result += "getTotalCharge() + ... + getFrequentRenterPoints();
    return result;
}
```

Outline of Today's Class

- Refactorings
  - We’ll learn about the State, Strategy and Template Method design patterns
  - Extract method, Move method
  - Replace temp with query
  - Replace Type Code with State/Strategy
  - Replace Conditional with Polymorphism
  - Form Template Method
  - Replace Magic Number with Symbolic Constant
- Refactoring, Conclusion

Code Smell: Long Method

- Long method
  - How long is too long? It depends…
  - More than 20 lines is usually too long. 10 or less
  - Why is this a problem
    - The longer the method, the more complex it is
      - the harder it is to understand and maintain
  - Example
    - Method statement() in Customer was too long
  - Fix: Extract Method refactoring

Refactoring, Discussion

- Key points
  - Small-step, semantic-preserving rewrites
  - Hand-in-hand with unit testing
- Advantages
  - Improves readability, maintainability
  - Often, inserts design patterns in code
- Disadvantages
  - Often, amounts to the reverse of an optimization
**Code Smell: Unnecessary Coupling**

- Unnecessary coupling (also, “feature envy”)
  - A method in one class heavily using fields or methods of another class (needs “information” from that other object)
- Why is this a problem
  - Harder to understand. Ideally, the “information expert” class performs the operation
- Example
  - getCharge() did not belong in Customer
- Fix: Move Method refactoring

**Code Smell: Temp Variables**

- Temp variables
  - Code uses poorly documented temporaries
- Why is this a problem
  - Uninformative, make code hard to understand
- Example
  - thisAmount, totalAmount temps
- Fix: Replace Temp with Query refactoring
  - Replacing temp thisAmount with a call to Rental.getCharge() improved readability

**Code Smell: Switch Statements**

- Switch statements
  - Code for different “instances” of same class
- Why is this a problem
  - Switch appears in many places. Hard to maintain
- Example
  - Switch on Movie._priceCode data
- Fix: Replace Type Code with State/Strategy, Replace Conditional with Polymorphism
  - Create abstract class and subclasses. Move each case-branch code into its subclass

**Code Smell: Duplicate Code**

- Duplicate code
  - Same or similar code appears in many places
- Why is this a problem
  - Often we fix a bug in one clone, but forget the others
- Example
  - TextStatement.value and HtmlStatement.value
- Fix: Form Template Method refactoring
  - Extract different code then pull similar code into a template method in the superclass. Create hooks in subclasses for differences

**Code Smell: Misplaced Field**

- Misplaced field
  - A field is used by another class more than the class where it is defined
- Why is this a problem
  - Creates unnecessary coupling between the two classes
- Fix: Move Field refactoring
  - Create a new field in the new class, and change all the field’s uses

**Code Smell: Oversized Class**

- Oversized class
  - A class doing work that should be done by two
- Why is this a problem
  - Just as with a method, the larger and more complex a class, the harder it is to understand, maintain and reuse
- Fix: Extract Class refactoring
  - Create a new class and move the relevant methods and fields to the new class
Exercise

- Go to:
  www.cs.rpi.edu/~milanova/csci2600/handouts/
  LibraryPatron.java

- Code smells?

- Suggest refactorings
  - Many "correct" refactorings