Refactoring, cont.

Announcements

- ERRATA in hw8.test starter code
  - I have switched `expected` and `actual` in all `assertEquals`. Sorry!
  - E.g., I have `assertEquals(parser.evaluate(), true);` but it should be `assertEquals(true, parser.evaluate());`
  - Switch arguments in your local tests

- HW8 due December 1st
  - First, refactor
  - Then implement NOT Expression
  - Last, implement Visitors: Evaluate, PrintInorder

- Check your grades
  - Quiz 1-9 in LMS
  - Exam 1-2 in LMS
  - HW 0-6 in Homework Server

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

Refactoring

- 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

```java
public String statement() {
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += …+String.valueOf(each.getCharge())+…+"\n";
    }
    result += …+getTotalCharge() + … +
                getTotalFrequentRenterPoints();
    return result;
}
```

- Added a Method

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

- Still refactoring… Back to getCharge

```java
double getCharge() { // now in Rental
    double result = 0;
    switch (getPriceCode()) { // Now, switch over OWN 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;
}
```

- Replacing Conditional Logic

- Problem: A switch statement on own data is bad. A switch statement on someone else’s data is worse

- First step towards getting rid of switch statement: move getCharge and getFrequentRenterPoints from class Rental to class Movie:

```java
class Rental { // replace with delegation
double getCharge() { // now in Movie
double result = 0;
    switch (getPriceCode()) { // Now, switch over OWN 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;
}
```

- Move Method
Replacing Conditional Logic

- 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
  - We need to add new algorithms/strategies easily
  - Placing the pricing algorithms/strategies in Movie will make Movie complex and inflexible
  - Have this giant switch statement

The Replace Type Code with State/Strategy Refactoring

- Replaced _priceCode (the type code) with Price _price (Strategy)
- Strategy and State are often interchangeable
- Goal: replace switch with subtype polymorphism!

The State Pattern

- A TCPState object has reference to enclosing TCPConnection object:
  ```java
  class TCPConnection {
    private TCPState state;
    public TCPConnection() {
      state = TCPClosed.getInstance(this);
    }
  }
  class TCPClosed extends TCPState {
    public void open() {
      // do work to open connection
      connection.state = TCPListing.getInstance(this);
    }
  }
  ```
  State classes are often Singletons
Replace Type Code with State/Strategy

- Add the new concrete Price classes
- In Movie: int _priceCode becomes Price _price
- Change Movie’s accessors to use _price

\[
\text{int getPriceCode() \{ return _price.getPriceCode(); \}}
\]

\[
\text{void setPriceCode(int arg) \{}
\]

\[
\text{switch (arg) \{}
\]

\[
\text{case REGULAR: _price = new Regular();}
\]

\[
\text{\ldots}
\]

Move Method getCharge() from Movie to Price

\[
\text{double getCharge(int daysRented) \{ // now in Price…}
\]

\[
\text{double result = 0;}
\]

\[
\text{switch (getPriceCode()) \{ // Note this stays the same!}
\]

\[
\text{case REGULAR: result += (daysRented - 2)*1.5;}
\]

\[
\text{break;}
\]

\[
\text{case NEW_RELEASE:}
\]

\[
\text{result += daysRented*3;}
\]

\[
\text{break;}
\]

\[
\text{case CHILDRENS:}
\]

\[
\text{result += 1.5;}
\]

\[
\text{if (daysRented>3) result += (daysRented-3)*1.5;}
\]

\[
\text{break;}
\]

\[
\text{\}}
\]

\[
\text{return result;}
\]

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
  - First, replace the type code with State/Strategy
  - Second, place each case branch into a subclass, add virtual call (e.g., _price.getCharge(daysRented))

Still Refactoring…

public String statement() { // in Customer
\[
\text{Iterator<Rental> rentals = _rentals.iterator();}
\]

\[
\text{String result = "\text{Rental Record for }" + getName() + "\n";}
\]

\[
\text{while (rentals.hasNext()) \{}
\]

\[
\text{Rental each = rentals.next();}
\]

\[
\text{result += \ldots + String.valueOf(each.getCharge()) + \ldots + "\n";}
\]

\[
\text{\}}
\]

\[
\text{result += \ldots + getTotalCharge() + \ldots + getTotalFreqRenterPoints() + \ldots}
\]

\[
\text{return result;}
\]

At some point, we created htmlStatement() which was the same, except that result had HTML symbols.
public String htmlStatement() { // in Customer
  Iterator<Rental> rentals = _rentals.iterator();
  String result = "<H1>Rental Record for <EM>"+
    getName() + "</EM><H1><P>
  while (rentals.hasNext()) {
    Rental each = rentals.next();
    result += ... + each.getCharge()+...+"n"; // + HTML
  }
  result +=... + getTotalCharge()... +
    getFrequentRenterPoints(); // + HTML
} // in Customer

Before we deal with duplicate code…

- Introduce Strategy for printing statements
  abstract class Statement {} // Abstract Strategy
  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 htmlStatment() { 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

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

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

Form Template Method step 1: Extract Method

class TextStatement extends Statement {
    public String value(Customer c) {
        Iterator<Rental> rentals = c.getRentals();
        String result = headerString(c);
        while (rentals.hasNext()) {
            Rental each = rentals.next();
            result += eachRentalString(each);
        }
        result += footerString(c);
        return result;
    }
}

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) {
        Iterator<Rental> rentals = c.getRentals();
        String result = headerString(c);
        while (rentals.hasNext()) {
            Rental each = rentals.next();
            result += eachRentalString(each);
        }
        result += footerString(c);
        return result;
    }
}
```

Form Template Method

- 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!

One last refactoring. Back to getCharge(int)

```
class Regular extends Price {
    double getCharge(int daysRented) {
        double result = 2;
        if (daysRented > 2) {
            result += (daysRented - 2)*1.5;
        }
        return result;
    }
}
```
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;
    }
}

---

**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: the Extract 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: the Replace Temp with Query refactoring
  - Replacing temp thisAmount with a call to Rental.getCharge() improved readability

---

**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: 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: the Form Template Method refactoring
  - Extract different code then pull similar code into a template method in the superclass. Create hooks in subclasses for differences

Refactoring

- Improves readability and understandability
  - Sometimes, at the expense of performance
- Key points: each small-step transformation immediately followed by testing
- Key point: intertwined with addition of code
  - Complex "ugly", but working! code + tests
  - While not done refactoring and coding
    - Refactor and test
    - Add more tests, then add more code

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

HAPPY THANKSGIVING!