

## Refactoring

## Announcements

- HW7 due today, HW8 coming up tomorrow (I'm taking a late day on posting HW8)
- Grades and feedback for HW0-5 in Homework Server
- Exam1-2, Quiz1-7 in LMS
- Quiz 9

2

## Outline of Today's Class

- Introduction to Refactoring
- Refactorings
  - 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

We'll learn about the **State, Strategy** and **Template Method** design patterns

3

## So Far

- We studied techniques for writing **correct** and **maintainable** code
  - Correctness: careful planning, specifications, reasoning about code, testing
  - Understandability and maintainability: Design patterns promote low coupling and "open/close" designs (i.e., designs that are "open for extension but closed to modification")

Fall 15 CSCI 2600, A Milanova

4

## So Far

- How to design your code
  - The hard way: Start hacking. Hack some more...
  - The easier way: Plan carefully
- How to verify your code
  - The hard way: Make up some inputs...
  - An easier way: Systematic testing and reasoning
- The hard way leads down the dark path
- But we do get down the dark path

Fall 15 CSCI 2600, A Milanova

5

## Refactoring

- Premise: we have written complex (ugly) code, but it works! Can we simplify this code?
- Refactoring: disciplined rewrite of code
  - Small-step behavior-preserving transformations
  - Followed by execution of test cases
- Continuous refactoring combined with testing is an essential software development practice

Fall 15 CSCI 2600, A Milanova

6

## Refactoring

- Refactorings attack **code smells**
- **Code smells** – bad coding practices
  - E.g., big method
  - An oversized “God” class
  - Similar methods, classes or subclasses
  - Little or no use of subtype polymorphism
  - High coupling between objects,
  - Etc.

Fall 15 CSCI 2600, A. Milanova

7

## Refactoring Activities

- Make long methods shorter
- Remove duplicate code
- Introduce design patterns
- Remove the use of hard-coded constants
- Etc...
- Goal: achieve code that is short, tight, **clear** and without duplication

Fall 15 CSCI 2600, A. Milanova

8

## A Refactoring

- A **refactoring** is a named, documented algorithm that attacks a specific bad coding practice
  - E.g., **Extract method**, **Move method**, **Replace constructor call with Factory method**
  - Relatively well-defined mechanics, can be automated
- Canonical Reference: Refactoring, Improving the Design of Existing Code by Martin Fowler
  - Initial catalog of 72 refactorings, 1999
  - Currently, more than 100 documented refactorings

Fall 15 CSCI 2600, A. Milanova

9

## Movie Rentals (Fowler, 1999)

```
public class Movie { // immutable class!
    public static final CHILDRENS = 2;
    public static final REGULAR = 0;
    public static final NEW_RELEASE = 1;
    private String _title;
    private int _priceCode;
    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }
    public String getTitle() { return _title; }
    public int getPriceCode() {
        return _priceCode;
    }
}
```

Fall 15 CSCI 2600, A. Milanova

10

## Class Rental

```
public class Rental { // an immutable class
    private Movie _movie;
    private int _daysRented;
    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    public Movie getMovie() { return _movie; }
    public int getDaysRented() {
        return _daysRented;
    }
}
```

Fall 15 CSCI 2600, A. Milanova

11

## Class Customer

```
public class Customer { // mutable class

    private String _name;
    private List<Rental> _rentals;

    public Customer (String name) {
        _name = name;
        _rentals = new ArrayList<Rental>();
    }
    public void addRental(Rental arg) {
        _rentals.add(arg);
    }
}
```

Fall 15 CSCI 2600, A. Milanova

12

## Method statement() in Customer, composes a Customer statement

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Iterator<Rental> rentals = _rentals.iterators();
    String result = "Rental Record for " + getName()+"\n";
    while (rentals.hasNext()) {
        // process current rental
        double thisAmount = 0; // amount for this rental
        Rental each = rentals.next();

        // Next, compute amount due for current rental (each)
    }
}
```

Fall 15 CSCI 2600, A Milanova

13

## Method statement()

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
            thisAmount += (each.getDaysRented()-2)*1.5;
        break;
    case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented()*3;
        break;
    case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented()-3)*1.5;
        break;
} // end of switch statement
```

14

## Method statement()

```
// add frequent renter points contributed by current rental
frequentRenterPoints++;
if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
    && each.getDaysRented() > 1)
    frequentRenterPoints++;
result += "\t" + each.getMovie().getTitle() + "\t" +
    String.valueOf(thisAmount) + "\n";
totalAmount += thisAmount;
} // end of the while loop over the _rentals
// add totalAmount and frequentRenterPoints to result String
return result; // Finally DONE!
} // end of method statement()
} // end of Customer class
```

15

## Discussion

- What code smells can you find?
- There are many...
- Code smells: big method, high coupling between objects, little or no use of subtype polymorphism, and so on

Fall 15 CSCI 2600, A Milanova

16

## The Extract Method Refactoring

- Problem: Method **Customer.statement()** is too big, difficult to understand and maintain
- Solution: Find a logical chunk of code to be extracted out of **statement()**, and perform the Extract Method refactoring
- Key point: **safety** – refactoring preserves behavior. Test after every refactoring!

What part of **statement()** would you extract?

Fall 15 CSCI 2600, A Milanova

17

## Extract Method, Mechanics

- Create a new method, name it appropriately
- Copy the extracted code in the new method
- Scan the extracted code for references to local variables
- If local variables are used only in extracted code, declare them in the new method
- See if any local variables are modified by the extracted code – tricky part...
- Pass as parameters local variables that are not modified but are only used by the extracted code
- Compile
- Replace the extracted code with a method call
- Compile and test!!!**

18

## Extract Method, extracted code

```
double thisAmount = 0; // thisAmount is local to while loop
switch (each.getMovie().getPriceCode()) { // each is local to while loop too
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
            thisAmount += (each.getDaysRented()-2)*1.5;
        break;
    case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented()*3;
        break;
    case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented()>3)
            thisAmount += (each.getDaysRented()-3)*1.5;
        break;
}
totalAmount += thisAmount;
```

19

## Extract Method. The new method

```
private double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented()-2)*1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented()*3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented()>3)
                thisAmount += (each.getDaysRented()-3)*1.5;
            break;
    }
    return thisAmount;
}
```

For readability, rename `thisAmount` to `result`

20

## Extract Method. The new method, still in class Customer

```
private double amountFor(Rental each) {
    double result = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (each.getDaysRented() > 2)
                result += (each.getDaysRented()-2)*1.5;
            break;
        case Movie.NEW_RELEASE:
            result += each.getDaysRented()*3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (each.getDaysRented()>3)
                result += (each.getDaysRented()-3)*1.5;
            break;
    }
    return result;
}
```

What's still "wrong" here?

21

## The Move Method Refactoring

- Problem: Unnecessary coupling from **Customer** to **Rental** through `getDaysRented()`. Unnecessary coupling to **Movie** too. **Customer** does not have the "information" to compute the rental amount
- Solution: Move `amountFor(Rental)` to the class that is the logical "information expert"
- Key point: **safety**. Test after refactoring!

What class has the information to compute the charge amount?

Fall 15 CSCI 2600, A Milanova

22

## Move Method, Mechanics

- Examine all features used by the source method that are defined on the source class. Consider if they should be moved also.
- Check the sub- and superclasses for other declarations of the method (virtuals).
- Declare the method in the target class.
- Appropriately copy the code from source to target.
- Compile the target class.
- Reference the correct target object from the source.
- Turn the source method into a delegating method.
- **Compile and test.**

23

## Move Method. `getCharge()`, now in class Rental

```
double getCharge() { // now in Rental!
    double result = 0;
    switch (getMovie().getPriceCode()) { // No reference to each!
        case Movie.REGULAR:
            result += 2;
            if (getDaysRented() > 2) // No each, Rental object has info
                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;
}
```

24

## Move Method

Initially, replace body of old method with delegation:

```
class Customer {  
    private double amountFor(Rental aRental) {  
        return aRental.getCharge();  
    }  
}
```

Compile and test to see if it works.

Next, find each reference to **amountFor** and replace with call to the new method:

**thisAmount = amountFor(each);** becomes

**thisAmount = each.getCharge();**

Compile and test!

25

## New and improved statement()

```
public String statement() {  
    double totalAmount = 0;  
    int frequentRenterPoints = 0;  
    Iterator<Rental> rentals = rentals.iterator();  
    String result = "Record for " + getName() + "\n";  
    while (rentals.hasNext()) {  
        double thisAmount = 0;  
        Rental each = rentals.next();  
        thisAmount = each.getCharge(); // BIG CHANGE!  
        // ...code for frequent renter points  
        result += "\t" + each.getMovie().getTitle() + "\t" +  
            String.valueOf(thisAmount) + "\n";  
        totalAmount += thisAmount;  
    }  
    // code to add totalAmount and  
    // frequentRenterPoints to result string, return result and DONE!  
}
```

Is **thisAmount** necessary?

26

## The Replace Temp with Query Refactoring

- Problem: Temporary variable **thisAmount** is meaningless, hinders readability
- Solution: Replace **thisAmount** with "query method" **each.getCharge()**
  - Claim: A "query method" is more informative
- Aside: "query methods" are free of side effects (i.e., they **modify** nothing)
- Key point: **safety**. Test after refactoring!

Fall 15 CSCI 2600, A Milanova

27

## Replace Temp With Query, Mechanics

- Look for a temporary variable that is assigned to only once. Why?
- Declare the temp as **final**
- **Compile** (makes sure temp is assigned once!)
- Extract the right-hand side of the assignment into a "query"; replace all occurrences of temp with query
  - Method computing the value of temp should be a "query" method, i.e., it should be free of side effects! Why?
- **Compile and test**

Fall 15 CSCI 2600, A Milanova

28

## Replace Temp with Query

```
public String statement() {  
    double totalAmount = 0;  
    int frequentRenterPoints = 0;  
    Iterator<Rental> rentals = rentals.iterator();  
    String result = "Record for " + getName() + "\n";  
    while (rentals.hasNext()) {  
        Rental each = rentals.next();  
        double thisAmount = each.getCharge();  
        // ...code for frequent renter points  
        result += "\t" + each.getMovie().getTitle() + "\t" +  
            String.valueOf(each.getCharge()) + "\n";  
        totalAmount += each.getCharge();  
    }  
    // code to add totalAmount and  
    // frequentRenterPoints to result string, return result and DONE!
```

We got rid of **thisAmount** temp, replaced it with **each.getCharge()**. Do you see issues with this refactoring?

29

## What else can we do?

```
public String statement() {  
    double totalAmount = 0;  
    int frequentRenterPoints = 0;  
    Iterator<Rental> rentals = rentals.iterator();  
    String result = "Record for " + getName() + "\n";  
    while (rentals.hasNext()) {  
        Rental each = rentals.next();  
        frequentRenterPoints++;  
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)  
            && each.getDaysRented() > 1)  
            frequentRenterPoints++;  
        result += "\t" + each.getMovie().getTitle() + "\t" +  
            String.valueOf(each.getCharge()) + "\n";  
        totalAmount += each.getCharge();  
    }  
    // end while  
    // code to add totalAmount and  
    // frequentRenterPoints to result string, return result, and DONE!  
}
```

30

## Extract Method + Move Method

```
public String statement() {
    ...
    while (...) { ...
        // add frequent renter and other bonus points:
        frequentRenterPoints++;
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
            && each.getDaysRented() > 1)
            frequentRenterPoints++;
        ...
    }
    // code to add totalAmount and frequentRenterPoints to result
}

After Extract Method & Move Method we have:
frequentRenterPoints += each.getFrequentRenterPoints();
```

31

## Replace Temp with Query, again

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        frequentRenterPoints += each.getFrequentRenterPoints();
        result += ...+String.valueOf(each.getCharge())+...+"\n";
        totalAmount += each.getCharge();
    }
    result += ... totalAmount...+...frequentRenterPoints+...
    return result;
}

Can we replace these last two temps?
```

Fall 15 CSCI 2600, A Milanova

32

## Replace Temp with Query

- Extract computation for **totalAmount** in a separate method in **Customer**. Is it a “query”?

```
private double getTotalCharge() {
    double result = 0;
    Iterator<Rental> rentals = _rentals.iterator();
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += each.getCharge();
    }
    return result;
}
```

Fall 15 CSCI 2600, A Milanova

33

## Replace Temp with Query

- Similarly, extract computation for **frequentRenterPoints**

```
private double getTotalFrequentRenterPoints() {
    double result = 0;
    Iterator<Rental> rentals = _rentals.iterator();
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += each.getFrequentRenterPoints();
    }
    return result;
}
```

Fall 15 CSCI 2600, A Milanova

34

## First, take totalAmount out

```
public String statement() {
    int frequentRenterPoints = 0; // first, take totalAmount out
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        frequentRenterPoints += each.getFrequentRenterPoints();
        result += ...+String.valueOf(each.getCharge())+...+"\n";
    }
    result += ...+getTotalCharge()+...+frequentRenterPoints
    return result;
}
```

Fall 15 CSCI 2600, A Milanova

The key point: small steps, preserving behavior!

35

## Next, take frequentRenterPoints out

```
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;
}

Methods getTotalCharge() and getTotalFrequentRenterPoints(),
two private methods in Customer. Both iterate over the
rentals. Issues?
```

36

## Refactoring So Far

- Small-step, behavior-preserving transformations. Continuously test.
- Goal: achieve code that is short, tight, clear and without duplication. Eliminate **code smells**
- Refactorings
  - Extract method
  - Move method
  - Replace temp with query... More

Fall 15 CSCI 2600, A.Milanova

37

## Now, let's add a method

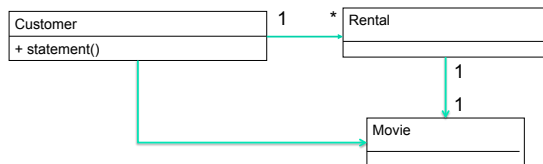
```
public String htmlStatement() {
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "<H1>Rental Record for <EM>" + getName() +
        "</EM><H1><P>\n";
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += ...+each.getCharge()+...+"\n"; // add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

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

38

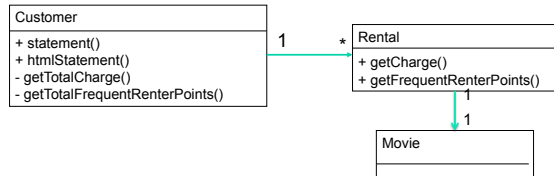
## Before...

- 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**



## Still refactoring... Back to getCharge

```
double getCharge() { // now in Rental
    double result = 0;
    switch (getMovie().getPriceCode()) {
        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;
}
```

What's wrong here?

41

## Replacing Conditional Logic

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

- First step towards solution: move `getCharge` and `getFrequentRenterPoints` from Rental to Movie:

```
class Rental { // replace with delegation
    double getCharge() {
        return _movie.getCharge(_daysRented);
    }
}
```

Fall 15 CSCI 2600, A.Milanova

42

## Move Method

```
double getCharge(int daysRented) { // now in Movie
    double result = 0;
    switch (getPriceCode()) { // Now, switch is on OWN data
    case Movie.REGULAR:
        result += 2;
        if (daysRented > 2) result += (daysRented - 2) * 1.5;
        break;
    case Movie.NEW_RELEASE:
        result += daysRented * 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 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()`

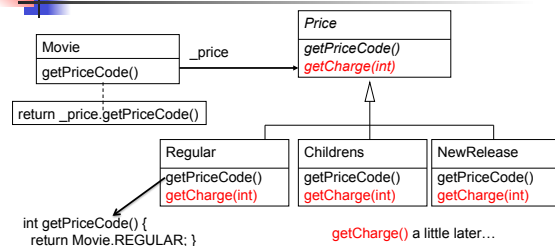
## The Strategy Design Pattern

- 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** too big and too complex
    - Switch statement code smell

Fall 15 CSCI 2600, A. Milanova

45

## The Replace Type Code with State/Strategy Refactoring



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

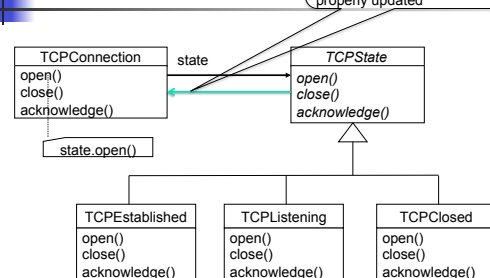
## Aside: the State Design Pattern

- 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

Fall 15 CSCI 2600, A. Milanova

47

## The State Pattern



Fall 15 CSCI 2600, A. Milanova

State classes are often Singletons

48



## The State Pattern

- A TCPState object has reference to enclosing TCPConnection object:

```
class TCPConnection {
    private TCPState state;
    public TCPConnection() {
        state = new TCPClosed(this);
    }
}
class TCPClosed extends TCPState {
    private TCPConnection connection;
    public void open() {
        // do work to open connection
        connection.state = new TCPListing();
    }
}
```

49

## 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`

```
int getPriceCode() { return _price.getPriceCode(); }
void setPriceCode(int arg) {
    switch (arg) {
        case REGULAR: _price = new Regular();
        ... }
}
```

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

## Move Method: `getCharge()` moves from **Movie** to **Price**

```
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 (getDaysRented > 3) result += (getDaysRented - 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 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)`)

53

