Refactoring

These slides are based on:

Chapter 1 of "Refactoring : improving the design of existing code, by Martin Fowler, Addison-Wesley, 1999"

1

What Is Refactoring

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
- It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
- In essence when you refactor you are improving the design of the code after it has been written.

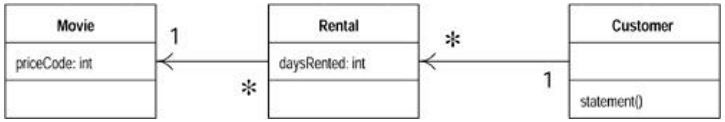
2

The example

- The program calculates and prints a statement of a customer's charges at a video store.
- The program is told which movies a customer rented and for how long.
- It then calculates the charges, which depend on how long the movie is rented, and the types of movies.
  - There are three kinds of movies: regular, children's, and new releases.
- In addition to calculating charges, the program also computes frequent renter points, which vary depending on whether the film is a new release.

3

Class diagram of the example



4

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

5

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

6

```
class Customer {
        private String _name;
        private Vector _rentals = new Vector();
        public Customer (String name){
                _name = name;
        };
        public void addRental(Rental arg) {
                _rentals.addElement(arg);
        }
        public String getName (){
                return _name;
        };
        public String statement() {
                double totalAmount = 0;
                int frequentRenterPoints = 0;
                Enumeration rentals = _rentals.elements();
                String result = "Rental Record for " + getName()
                                + "\n";
```

7

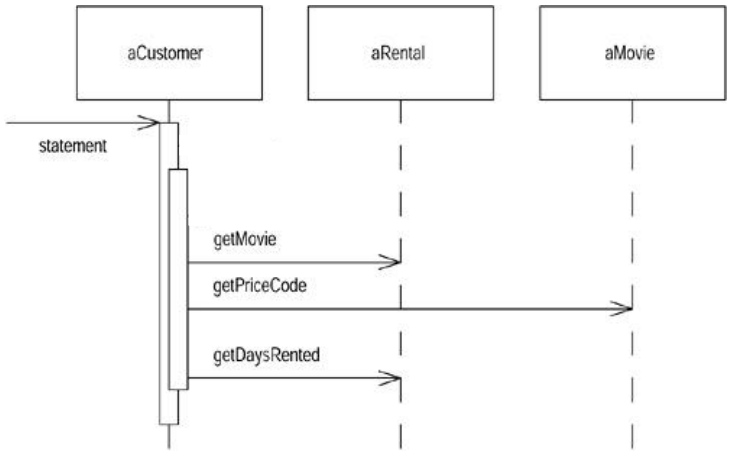
```
while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 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;
        }
```

8

```
    // add frequent renter points
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
        &&  each.getDaysRented() > 1)
        frequentRenterPoints ++;
    //show figures for this rental
    result += "\t" + each.getMovie().getTitle()+ "\t" +
              String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }
  //add footer lines
  result += "Amount owed is " + String.valueOf(totalAmount) +
            "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints)
            + " frequent renter points";
  return result;
}
```

9

Interactions for the statement method



Comments on the Program

- The users want a statement printed in HTML so that the statement can be Web enabled
  - It is impossible to reuse any of the behavior of the current statement method for an HTML statement

```
Rental Record for x
Lion King 1.5
Star Wars 3.5
Amount owed is 5.0
You earned 2 frequent renter points
```

```
<H1>Rentals for <EM>x</EM></H1><P>
Lion King: 1.5<BR>
Star Wars: 3.5<BR>
<P>You owe <EM>5.0</EM><P>
On this rental you earned <EM>2</EM> frequent renter points<P>
```

- You write a whole new method, htmlStatement, that duplicates much of the behavior of method statement.
  - You can just copy the statement method and make whatever changes you need.
- What happens when the charging rules change?
  - You have to fix both methods statement and htmlStatement to ensure the fixes are consistent.

12

- What if the users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make.
  - These changes will affect both the way renters are charged for movies and the way that frequent renter points are calculated.
- In reality, whatever scheme users come up with, the only guarantee you're going to have is that they will change it again within six months.

13

- The statement method is where the changes have to be made to deal with changes in classification and charging rules.
- If, however, we copy method statement to an method htmlStatement, we need to ensure that any changes are completely consistent.
- Furthermore, as the rules grow in complexity, it's going to be harder to figure out where to make the changes and harder to make them without making a mistake.
- We need to factor the program to make it easier to add features to the program

14

First Step in Refactoring

- Create a test suite for testing the section of the code that will be refactored.
- This would ensure that the refactored code still provide the same functionality as the old code
- Automate the tests if possible
  - This allows the test to be done easily and quickly
  - JUnit

15

Decomposing and Redistributing the Statement Method

- The statement method is overly long.
- Decompose long method into smaller pieces.
  - Smaller pieces of code tend to make things more manageable.
  - They are easier to work with and move around.
- Our aim is to make it easier to write an htmlStatement method with much less duplication of code.
  - statement and htmlStatement methods both need calculate the cost and the frequent renter points
  - They are different in presenting the results
- Find a logical clump of code and create a method out of it.
  - An obvious piece here is the switch statement.
  - It calculates the cost. So, it can be used by both statement and htmlStatement methods

16

```
while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        // determine amounts for each line
        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;
        }
```

17

- First look in the fragment for any variables that are local in scope to the method we are creating, and parameters.
- This segment of code uses two variables: each and thisAmount.
- each is not modified but thisAmount is modified.
- Any nonmodified variable can be passed in as a parameter.
- Modified variables need more care. thisAmount is initialized to 0 each time around the loop and is not altered until the switch gets to it.
  - So we can just assign the result to thisAmount and return the value of it from the method.

18

```
public String statement() {
        ...
        thisAmount = amountFor(each);
        ...
```

```
private double amountFor(Rental each) {
        double thisAmount = 0;
        switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
                thisAmount += 2;
                ...
        case Movie.NEW_RELEASE:
                ...
        case Movie.CHILDRENS:
                ...
        }
        return thisAmount;
}
```

19

rename code

- Give meaningful names to variables
  - Improve the maintainability of the program
- Good code should communicate what it is doing clearly, and variable names are a key to clear code.
- Good programmers write code that humans can understand.

20

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

21

Move Method

- amountFor method in Customer uses information from the rental object, but does not use information from the Customer.
- In most cases a method should be on the object whose data it uses, thus the method should be moved to Rental.
- Copy the code over to Rental, adjust it to fit in its new home
  - remove the parameter
  - rename the method
- Replace the body of Customer.amountFor to delegate to the new method in Rental

22

class Customer

```
public String statement() {
        ...
        while (rentals.hasMoreElements()) {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();
                thisAmount = amountFor(each);
        ...

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

23

class Rental

```
double getCharge() {
        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;
}
```

24

- The amountFor in Customer simply delegates the call to Rental's getCharge method.
- We can call Rental's getCharge method directly.
- The next step is to find every reference to the old method and adjust the reference to use the new method.

25

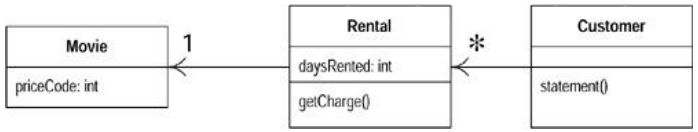
class Customer

```
public String statement() {
        ...
        while (rentals.hasMoreElements()) {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();
                //thisAmount = amountFor(each);
                thisAmount = each.getCharge();
        ...

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

26

Class diagram after moving the method



27

Replace temporary variables with query

- thisAmount is set to the result of each.charge and not changed afterward.
- thisAmount can be eliminated
- Temporary variables are often a problem in that you can easily lose track of what they are there for.
- Of course there is a performance price to pay
  - the charge is now calculated twice.
- The secret to fast software is to write tuneable software first and then to tune it for sufficient speed.

28

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration<Rental> rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();
                thisAmount = each.getCharge();
                …
                // show figures for this rental
                result += "\t" + each.getMovie().getTitle() + "\t"
                                + String.valueOf(thisAmount) + "\n";
                totalAmount += thisAmount;
        }
        …
}
```

29

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration<Rental> rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
                Rental each = (Rental) rentals.nextElement();
                …
                // show figures for this rental
                result += "\t" + each.getMovie().getTitle() + "\t"
                        + String.valueOf(each.getCharge()) + "\n";
                totalAmount += each.getCharge();
        }
        …
}
```

30

Extracting Frequent Renter Points Calculation

- Frequent renter points calculation is needed by both the statement and the htmlStatement method.
- We can refactor it as we did to the cost calculation

31

public String statement()

```
int frequentRenterPoints = 0;
…
while (rentals.hasMoreElements()) {
   Rental each = (Rental) rentals.nextElement();
   // add frequent renter points
   frequentRenterPoints ++;
   // add bonus for a two day new release rental
   if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
         && each.getDaysRented() > 1)
         frequentRenterPoints ++;
   …
}
```

32

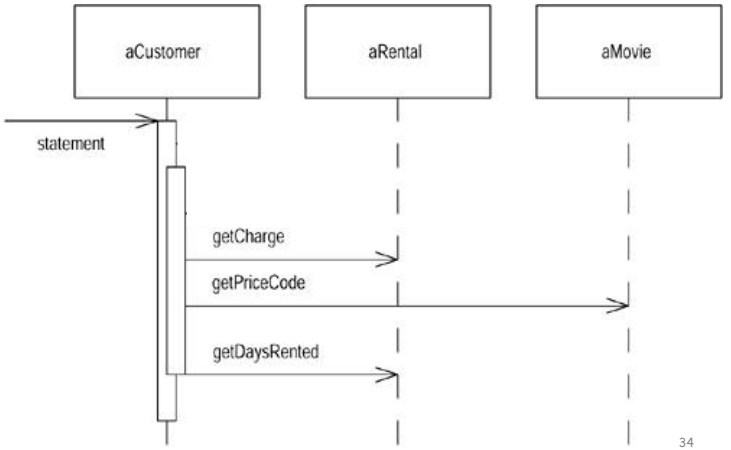
```
public String statement() {
        ...
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
        ...
    }
```

class Rental...

```
int getFrequentRenterPoints() {
        if ((getMovie().getPriceCode() == Movie.NEW_RELEASE)
            && getDaysRented() > 1)
                return 2;
        else
                return 1;
}
```

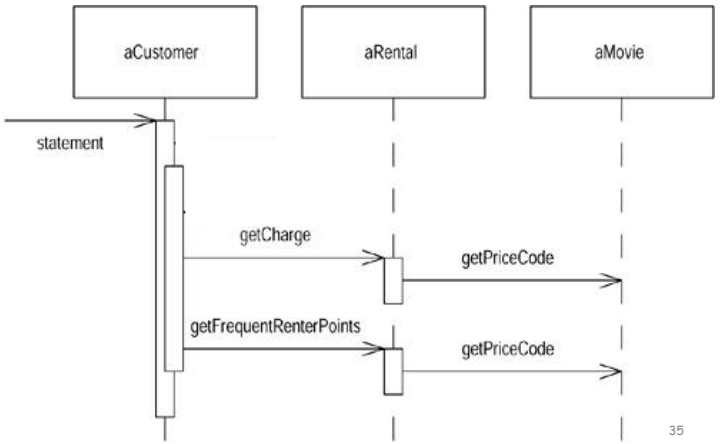
33

Sequence diagram before extraction and movement of the frequent renter points calculation



34

Sequence diagram after extraction and movement of the frequent renter points calculation



35

Removing Temps

- The total cost and the frequent renter points are needed by both statement and htmlStatement.
- They are currently being calculated in a loop.
- htmlStatement can copy the same loop in statement
  - What if the GST rate for different cost is different? E.g. cost above $20 is at 20% while the others are at 15%.
  - What if the frequent renter points policy is different? E.g. if you earn more than 10 points in one borrowing, you get 10 bonus points.
  - You have to change both the statement and htmlStatement method

36

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration<Rental> rentals = _rentals.elements();
   String result = "Rental Record for " + getName() + "\n";
   while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();
      result += "\t" + each.getMovie().getTitle() + "\t"
                + String.valueOf(each.getCharge()) + "\n";
      totalAmount += each.getCharge();
   }
   result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
   result += "You earned " + String.valueOf(frequentRenterPoints)
             + " frequent renter points";
   return result;
}
```

37

```
private double getTotalCharge() {
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       result += each.getCharge();
    }
    return result;
}
```

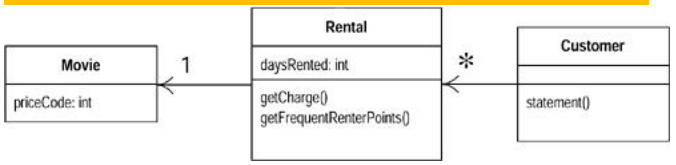
```
private int getTotalFrequentRenterPoints(){
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
       Rental each = (Rental) rentals.nextElement();
       result += each.getFrequentRenterPoints();
    }
    return result;
}
```

38

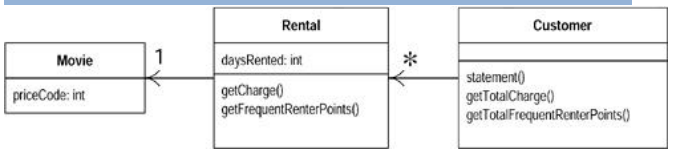
```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
    }
    result += "Amount owed is " +
    String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " +
    String.valueOf(getTotalFrequentRenterPoints()) +
                  " frequent renter points";
    return result;
}
```

39

Class diagram before removing frequentRenterPoints and totalAmount



Class diagram after removing frequentRenterPoints and totalAmount



40

Adding the htmlStatement method

- By extracting the calculations, we can create the htmlStatement method and reuse all of the calculation code that was in the original statement method.
- We don't copy and paste, so if the calculation rules change we have only one place in the code to go to (i.e. the method for carrying out the calculation).

41

```
public String htmlStatement() {
    Enumeration<Rental> rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + getName() +
                    "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getMovie().getTitle() + ": "
                  + String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += "<P>You owe <EM>" + String.valueOf(getTotalCharge())
              + "</EM><P>\n";
    result += "On this rental you earned <EM>"
              + String.valueOf(getTotalFrequentRenterPoints())
              + "</EM> frequent renter points<P>";
    return result;
}
```

42

Some general guidelines for refactoring

- Duplicated Code
  - If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.
  - eliminate this duplication by using Extract Method, i.e. creating new method
- Long Method
  - A long procedure is difficult to understand.
  - If you feel the need to comment something, you might want to write a method instead.
  - If you have a method with lots of temporary variables, Replace Temp with Query to eliminate the temporary variables.

43

- Long Parameter List
  - long parameter lists are hard to understand, because they become inconsistent and difficult to use,
  - if you don't have something you need, you can always ask another object to get it for you.
- Feature Envy
  - A method that seems more interested in a class other than the one it actually is in.
  - The most common envy is the data.
  - You move the method to the class where the data reside.
  - Often a method uses data of several classes, so which one should it live with?
    - The heuristic we use is to determine which class has most of the data and put the method with that data.
    - Or, break the method into smaller methods with each method only uses data from one class

44

review

- Refactor the program that you have write for the three assignments in this course
- Understand the purpose of refactoring.
- Understand the general guidelines for doing refactoring.

45