Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

What is refactoring?

- A change made to the internal structure of software to make it easier to understand and modify without changing its observable behavior
- Or, to restructure software by applying a series of refactoring without changing its observable behavior

Why Refactoring?

- Improves the design of software
  - The design of a program will inevitably decay, and refactoring helps to restore its structure
- Makes software easier to understand
  - When we program, we often focus on how to make it work, i.e., understandable to the computer. Refactoring helps us to think about how to make it understandable to the people
- Helps to find bugs
  - Refactoring forces one to think deep about the program and thus help to detect bugs that may exist in the code
- Helps to code faster
  - A good design is essential for rapid software development

When to Refactor?

- Refactor when you add function
  - Refactor as you try to understand the code
  - "If only I'd designed the code this way, adding this feature would be easy."
- Refactor when you need to fix a bug
  - The very existence of a bug might indicate that the code is not well-structured
- Refactor when you do a code review
  - Refactoring helps to produce concrete results from code review

Problems with Refactoring

- Refactoring may change interfaces; this needs to be handled carefully
  - Retain the old interface until the users have had a chance to react to the change
- Some designs may be very difficult to change
  - When a design choice is to be made, consider how difficult it would be if later you have to change it.
  - If it seems easy, then don't worry too much about the choice. Otherwise, be careful.

When not to refactor?

- Easier to write from scratch
  - The current code just doesn't work
- Close to a deadline
  - The long term benefit of refactoring may not appear until after the deadline

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

A Video Rental System (1)

```
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;
    }
}
```

A Video Rental System (2)

```
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;
    }
}
```

A Video Rental System (3)

```
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 () {
        …
    }
}
```

A Video Rental System (4)

```
public String statement () {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements ();
   String result = "Rental Record for " + getName () + "\n";
   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.CHILDREN:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
               thisAmount += (each.getDaysRented() – 3) * 1.5;
            break;
      }

      // add frequent renter points
      frequentRenterPoints ++;
      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;
}
```

Good or Bad?

- What is your opinion?
- The code does not seem to be balanced.
- What if we need to add a method to print statement in HTML?

First Step in Refactoring

- Build a solid set of tests for the code you plan to refactor.
- For example, we can create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings.

Move the Amount Calculation

```
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.CHILDREN:
          result += 1.5;
          if (getDaysRented() > 3)
            result += (getDaysRented() – 3) * 1.5;
          break;
    }
    return result;
}
```

Move the Amount Calculation

```
public String statement () {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement ();

        // add frequent renter points
        frequentRenterPoints ++;
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1)
            frequentRenterPoints ++;

       // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t" + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
                    + " frequent renter   points";
    return result;
}
```

Extracting Frequent Renter Points

```
public String statement () {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        frequentRenterPoints += each.getFrequentRenterPoints ();

       // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t"
                    + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
                  + " frequent renter   points";
    return result;
}

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

Extracting Total Amount Calculation

```
public String statement () {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements ();
    String result = "Rental Record for " + getName () + "\n";
    while (rentals.hasMoreElements ()) {
        Rental each = (Rental) rentals.nextElement ();
        frequentRenterPoints += each.getFrequentRenterPoints ();

       // show figures for this rental
        result += "\t" + each.getMovie.getTitle() + "\t"
                    + String.valueOf(each.getCharge()) + "\n";
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints)
                    + " frequent renter   points";
    return result;
}

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

Extract Total Frequent Renter Points Calculation

```
public String statement () {
    Enumeration 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";
    }
    // add footer lines
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(getTotalFrequentRenterPoints())
                + " frequent renter   points";
    return result;
}

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

Add htmlStatement

```
public String htmlStatement () {
   Enumeration rentals = _rentals.elements ();
   String result = "<H1>Rental Record for <EM>" + getName () + "</EM></
H1><P>\n";
   while (rentals.hasMoreElements ()) {
      Rental each = (Rental) rentals.nextElement ();

      // show figures for this rental
      result += each.getMovie.getTitle() + ": "
                  + String.valueOf(each.getCharge()) + "<BR>\n";
   }
   // add footer lines
   result += "<P>You owe <EM> " + String.valueOf(getTotalCharge())
                  + "</EM><P>\n";
   result += "On this rental you earned "
                  + String.valueOf(getTotalFrequentRenterPoints())
                  + " </EM>frequent renter points <P>";
   return result;
}
```

Switch Statement

- If we use a switch statement in a class A, we want to switch on the data members of the same class, not the data members of a different class B.
- Otherwise, if the data members of B change, we may forget to change the switch statement in A

Move getCharge() (2)

```
class Movie …
   double getCharge (int daysRented) {
      double result = 0;
      switch (getPriceCode ()) {
            case Movie.REGULAR:
               result += 2;
               if (daysRented () > 2)
                  result += (daysRented – 2) * 1.5;
               break;
            case Movie.NEW_RELEASE:
               result += daysRented * 3; break;
            case Movie.CHILDREN:
               result += 1.5;
               if (daysRented > 3)
                   result += (daysRented – 3) * 1.5;
               break;
      }
      return result;
}

class Rental …
   double getCharge () {
      return _movie.getCharge (_daysRented).
   }
```

Move getFrequentRenterPoints() (1)

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

Move getFrequentRenterPoints() (2)

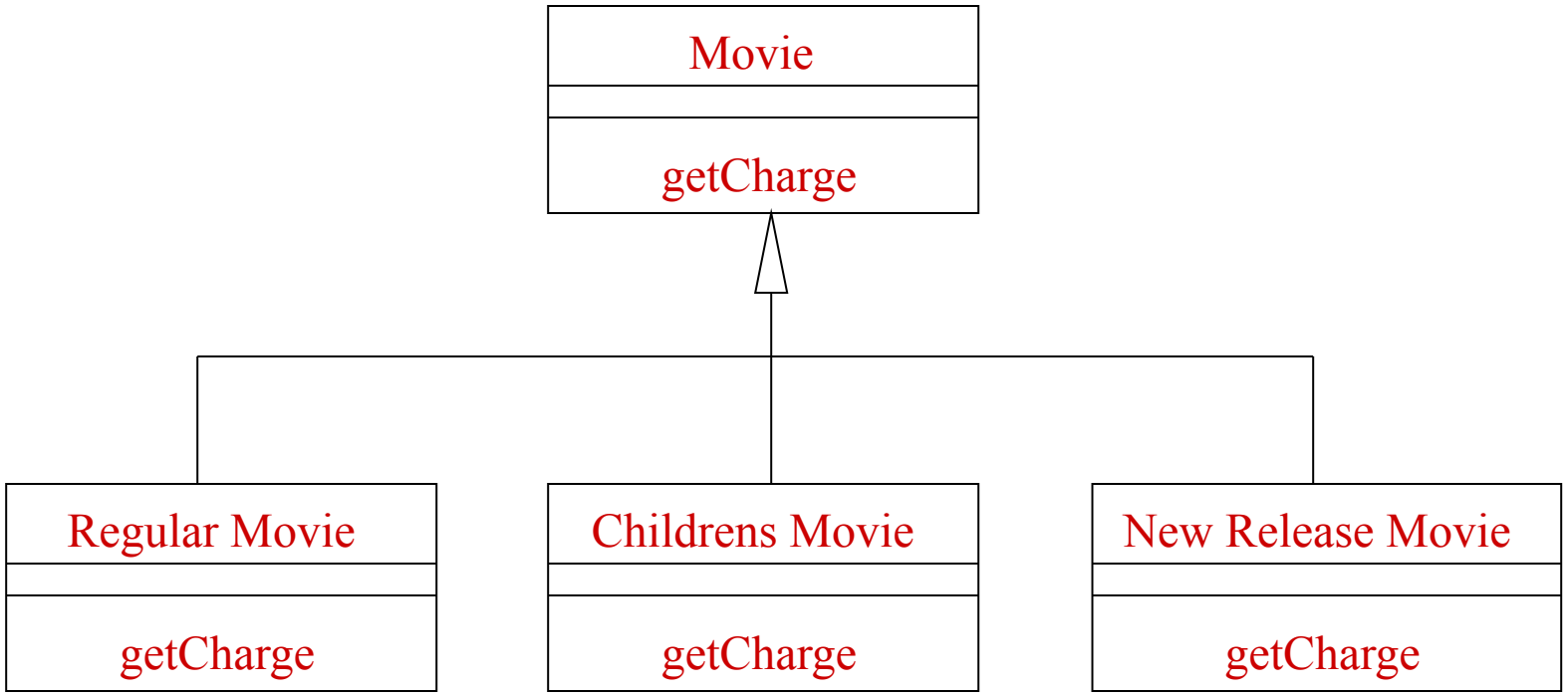
```
class Movie …
    int getFrequentRenterPoints () {
        if ((getPriceCode() == Movie.NEW_RELEASE)
                && each.getDaysRented() > 1)
          return 2;
        else
          return 1;
    }

class Rental …
    int getFrequentRenterPoints () {
        return _movie.getFrequentRenterPoints (_daysRent);
    }
```

Conditional vs Polymorphism

```
class Movie …
   double getCharge (int daysRented) {
      double result = 0;
      switch (getPriceCode ()) {
            case Movie.REGULAR:
               result += 2;
               if (daysRented () > 2)
                  result += (daysRented – 2) * 1.5;
               break;
            case Movie.NEW_RELEASE:
               result += daysRented * 3; break;
            case Movie.CHILDREN:
               result += 1.5;
               if (daysRented > 3)
                   result += (daysRented – 3) * 1.5;
               break;
      }
      return result;
}
```

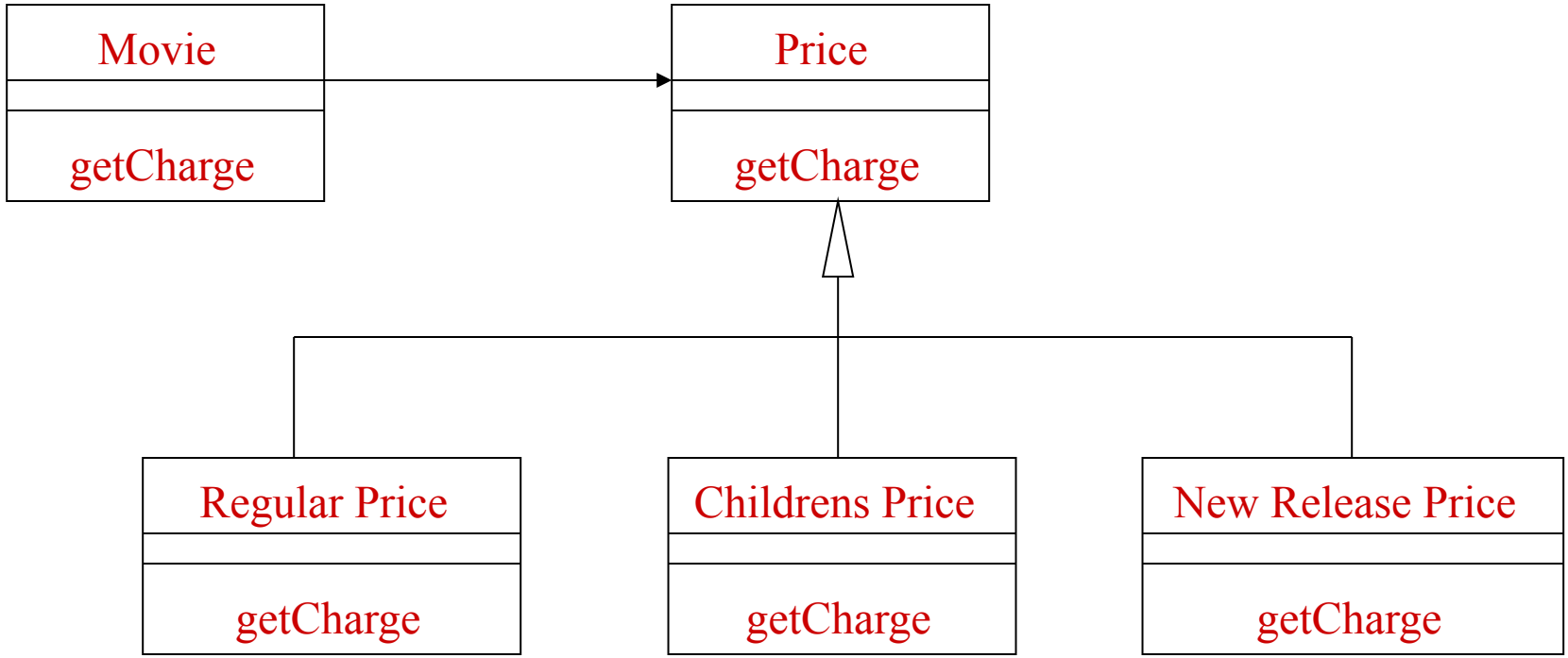
Conditional vs Polymorphism (2)



Can we do better?

- What happens if a New Release movie becomes a Regular movie after three months?
- Can we directly change a New Release movie object to a Regular movie object?
- Instead we have to create a new Regular movie object that is the same as the New Release movie object, except that the movie type is Movie_REGULAR.

Can we do better? (2)



Now we only need to change the association of the Price object from a New Release Price to a Regular Price.

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

Duplicated Code

- The same code structure appears in more than one place
  - Two methods of the same class, two sibling subclasses, or two unrelated classes
- Makes the code unnecessarily long, and also makes it difficult to maintain consistency
- Extract the common code and then invoke it from different places

Long Method

- The body of a method contains an excessive number of statements
- The longer a method is, the more difficult it is to understand and maintain
- Find parts of the method that seem to go nicely together and make a new method

Large Class

- Too many instance variables or too much code in the class
- Difficult to understand and maintain
- Break it up into several smaller classes, e.g., based on cohesion or how clients use the class

Long Parameter List

- A method that takes too many parameters in its signature
- Difficult to understand and use, and makes the interface less stable
- Use object to obtain the data, instead of directly pass them around

Feature Envy

- A method that seems more interested in a class other than the one it actually is in
- May not be in sync with the data it operates on
- Move the method to the class which has the most data needed by the method

Speculative Generality

- Excessive support for generality that is not really needed
- Make the code unnecessarily complex and hard to maintain
- Remove those additional support, e.g., remove unnecessary abstract classes, delegation, and parameters

Data Classes

- Classes that are dumb data holders, i.e., they only have fields and get/set methods
- These classes are often manipulated in far too much detail by other classes
- Give more responsibility to these classes

Comments

- Comments are often used as a deodorant: they are there because the code is bad
- Try to remove the comments by refactoring, e.g., extract a block of code as a separate method

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Summary

Summary

- Refactoring does not fix any bug or add any new feature. Instead, it is aimed to facilitate future changes.
- Refactoring allows us to do a reasonable, instead of perfect, design, and then improve the design later.
- In general, computation should be separate from presentation. This allows the same computation to be reused in different platforms.
- Bad smells signal opportunities for refactoring.