UTA

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Catalog of Refactorings
- Summary

UTA

What is refactoring?

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

UTA

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

UTA

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

UTA

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.

UTA

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

UTA

Refactoring and Design

- Upfront design: Do all the design up front, and try to find the best solution
- No design: Code first, get it working, and then refactor it into shape
- Combined: Still do upfront design, but only try to find a reasonable solution. Later, do refactoring to improve the design

UTA

Refactoring and Performance

- Does refactoring make software run slower?
- True in some cases. But in general, refactoring can make software more amenable to performance tuning.
- Three general approaches to write fast software
  - Time budgeting – typically applied in hard real-time environments only
  - Constant attention – slows development, and improvement may be made with a narrow perspective
  - Performance tuning – do performance tuning at a later stage of development

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Catalog of Refactorings
- 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;
    }
}
```

UTA

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;
   }
}
```

UTA

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 () {
      …
   }
}
```

UTA

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;
}
```

UTA

Good or Bad?

- What is your opinion?
- Consider the following possible changes:
  - Add a method to print statement in HTML
  - Make changes to the way the movies are classified, which could affect both the way renters are charged and the way frequent renter points are calculated

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.

Decompose statement

```
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 ();

      thisAmount  =  amountFor(each);

      // 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;
}

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

UTA

Rename Variables

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

UTA

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;
}

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

UTA

Remove the Old method

```
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 ();

      thisAmount = each.getCharge ();

      // 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;
}
```

UTA

Replace Temp with Query

```
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 ();

      // 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;
}
```

UTA

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;
   }
```

UTA

Remove Temps - totalAmount

```
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;
}
```

UTA

Remove Temps - frequentRenterPoints

```
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;
}
```

UTA

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;
}
```

UTA

Move getCharge() (1)

```
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;
}
```

UTA

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).
  }
```

UTA

Move getFrequentRenterPoints() (1)

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

UTA

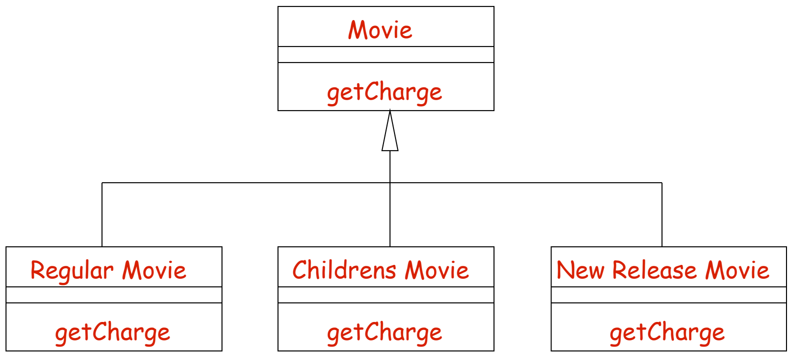
Move getFrequentRenterPoints() (2)

```
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);
   }
```

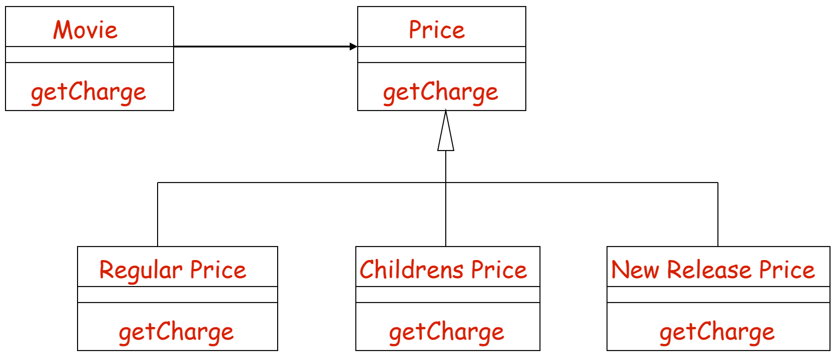
UTA

Replace Conditional with Polymorphism (1)



UTA

Replace Conditional with Polymorphism (2)



Self Encapsulating Field

```
class Movie…
   public Movie (String name, int priceCode) {
      _title = name;
      _priceCode = priceCode;
   }

class Movie…
   public Movie (String name, int priceCode) {
      _title = name;
      setPriceCode (priceCode);
   }
```

class Price

```
abstract class Price {
  abstract int getPriceCode ();
}
class ChildrensPrice extends Price {
  int getPriceCode () {
    return Movie.CHILDRENS;
  }
}
class NewReleasePrice extends Price {
  int getPriceCode () {
    return Movie.NEW_RELEASE;
  }
}
class RegularPrice extends Price {
  int getPriceCode () {
    return Movie.REGULAR;
  }
}
```

UTA

Change accessors (1)

```
public int getPriceCode () {
   return _priceCode;
}
public void setPriceCode (int arg) {
  _priceCode = arg;
}
private int _priceCode;
```

UTA

Change accessors (2)

```
class Movie…
  public int getPriceCode () {
    return _price.getPriceCode ();
  }
  public void setPriceCode (int arg) {
    switch (arg) {
       case REGULAR:
         _price = new RegularPrice (); break;
       case CHILDRENS:
         _price = new ChildrensPrice (); break;
       case NEW_RELEASE:
         _price = new NewReleasePrice (); break;
       default:
         throw new IllegalArgumentException("Incorrect Price Code");
    }

  private Price = _price;
```

UTA

Move getCharge() (1)

```
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;
}
```

UTA

Move getCharge() (2)

```
class Movie …
   double getCharge (int daysRented) {
       return _price.getCharge (daysRented);
   }

class Price …
   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;
}
```

UTA

Replace Conditional with Polymorphism

```
class RegularPrice ...
   double getCharge (int daysRented) {
      double result = 2;
      if (daysRented > 2)
         result += (daysRented – 2) * 1.5;
   }

class ChildrensPrice ...
   double getCharge (int daysRented) {
       double result = 1.5;
       if (daysRented > 3)
          result += (daysRented – 3) * 1.5;
       return result;
   }

class NewReleasePrice ...
   double getCharge (int daysRented) {
       return daysRented * 3;
   }

class Price ...
   abstract double getCharge (int daysRented);
```

UTA

getFrequentRenterPoints (1)

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

UTA

getFrequentRenterPoints (2)

```
class Movie ...
   int getFrequentRenterPoints (int daysRented) {
        return _price.getFrequentRenterPoints (daysRented);
   }

class Price ...
   int getFrequentRenterPoints (int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
   }

class NewReleasePrice
   int getFrequentRenterPoints (int daysRented) {
        return (daysRented > 1) ? 2 : 1;
   }

class Price...
   int getFrequentRenterPoints (int daysRented) {
        return 1;
   }
```

UTA

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Catalog of Refactorings
- Summary

UTA

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

UTA

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

UTA

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

UTA

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

Divergent Change/Shotgun Surgery

- ❑ Divergent change: a class is commonly changed in different ways for different reasons
  - Signals a low degree of cohesion
- ❑ Shotgun surgery: a change involves too many classes
  - Hard to identify all the classes, and some may get missed
- ❑ Ideally, we want to make arrangements so that there is one-to-one link between common changes and classes.

Put things together that change together!

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

UTA

Switch Statements

- The same switch statement scattered about a program in different places
- Adding a new clause requires changing all these duplicates
- Use polymorphism to replace the switch statements

UTA

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

UTA

Temporary Field

- Instance variables that are merely used for passing parameters
- Create unnecessary confusion as they are not an integral part of the object
- Create a new object to enclose those parameters and then pass the object around

UTA

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

UTA

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

UTA

Software Refactoring

- Introduction
- A Motivating Example
- Bad Smells in Code
- Catalog of Refactorings
- Summary

UTA

Format

- Name: the name of the refactoring
- Summary: a brief description about the refactoring
- Motivation: why the refactoring should be done and in what circumstances it shouldn't be done
- Mechanics: a concise, step-by-step description of how to carry out the refactoring
- Examples: a very simple use of the refactoring to illustrate how it works

UTA

Major Groups of Refactorings

- Composing methods
- Moving features between objects
- Simplifying conditional expressions
- Organizing data
- Making method calls simpler
- Dealing with generalization

Composing Methods (1)

- Extract Method
  - Turn a code fragment into a method with a meaningful name
- Inline Method
  - Put a method that is really simple into the body of its callers and remove the method
- Inline Temp
  - Replace a temp that is only assigned once with the actual expression
- Replace Temp with Query
  - Replace a temp with a query method
- Introduce Explaining Variable
  - Put intermediate results into a temp with a meaningful name

Composing Methods (2)

- Split Temporary Variable
  - Split a temp that is assigned more than once into several temps, one for each assignment
- Remove Assignments to Parameters
  - Parameters should not be assigned; use a temp instead.
- Replace Method with Method Object
  - Turn a very long method into an object, with all the local vars becoming fields on the object. If needed, you can decompose this method into several smaller ones on the object
- Substitute Algorithm
  - Replace the body of a method with a new algorithm

UTA

Extract Method (1)

- Need to consider how to deal with local variables:
  - No local variables
  - Read-only variables: Simply pass the values of those variables as parameters to the new method
  - Variables that may be changed
    - Only one variable is changed – return the new value from the new method
    - Multiple variables are changed – try to extract multiple methods, one for each variable

UTA

Extract Method (2)

```
void printOwing () {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;
   print Banner ();

   // calculate outstanding
   while (e.hasMoreElements ()) {
      Order each = (Order) e.nextElement ();
      outstanding += each.getAmount ();
   }

   printDetails (outstanding);
}
```

UTA

Extract Method (3)

```
void printOwing () {
    print Banner ();
    double outstanding = getOutstanding ();
    printDetails (outstanding);
}

Double getOutstanding () {
    Enumeration e = _orders.elements ();
    double result = 0.0;

    while (e.hasMoreElements ()) {
        Order each = (Order) e.nextElement ();
        result += each.getAmount ();
    }
    return result;
}
```

UTA

Replace Temp with Query

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

⇩

```
if (basePrice > 1000)
    return basePrice () * 0.95;
else
    return basePrice () * 0.98;
…
double basePrice () {
    return _quantity * _itemPrice;
}
```

UTA

Split Temporary Variable

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

⇩

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (are);
```

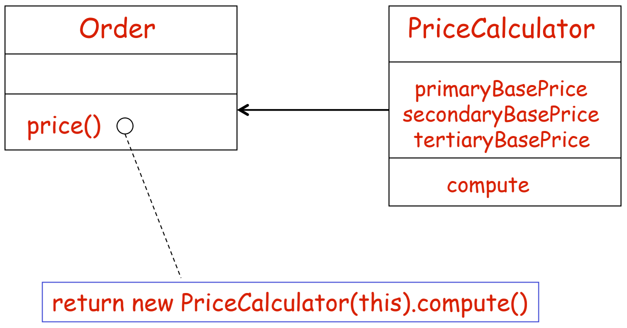
UTA

Replace Method with Method Object (1)

```
class Order …
  double price () {
      double primaryBasePrice;
      double secondaryBasePrice;
      double tertiaryBasePrice;
      // long computation
  }
```

UTA

Replace Method with Method Object (2)



UTA

Moving Features Between Objects (1)

- Move Method
  - Move a method from one class to another
- Move Field
  - Move a field from one class to another
- Extract Class
  - Extract a new class, with the relevant fields and methods, from a class that is doing too much work,
- Inline Class
  - Make a class that isn't doing much a part of another class

UTA

Moving Features Between Objects (2)

- Hide Delegate
  - Create methods on a server class to hide its delegate(s) from its client
- Remove Middle Man
  - Remove a class that is doing too much simple delegation
- Introduce Foreign Method
  - Create a method in a client class to implement what is needed from a server class but is not provided
- Introduce Local Extension
  - Create a wrapper or subclass of a server class to provide additional methods that are needed by a client class

UTA

Extract Class (1)

```
class Person ...
   public String getName () {
      return _name;
   }
   public String getTelephoneNumber () {
      return ("(" + _officeAreaCode + ") " + _officeNumber);
   }
  String getOfficeAreaCode () {
      return _officeAreaCode;
   }
   void setOfficeAreaCode (String arg) {
      _officeAreaCode = arg;
   }
  String getOfficeNumber () {
      return _officeNumber;
   }
   void setOfficeNumber (String arg) {
      _officeNumber = arg;
   }

   private String _name;
   private String _officeAreaCode;
   private String _officeNumber;
```

UTA

Extract Class (2)

```
class Person ...
  public String getName () {
    return _name;
  }
  public String getTelephoneNumber () {
    return ("(" + getOfficeAreaCode () + ") " + _officeNumber);
  }
  String getOfficeAreaCode () {
    return _officeTelephone.getAreaCode ();
  }
  void setOfficeAreaCode (String arg) {
    _officeTelephone.setAreaCode (arg);
  }
  String getOfficeNumber () {
    return _officeTelephone.getNumber ();
  }
  void setOfficeNumber (String arg) {
    _officeTelephone.setNumber (arg);
  }

  private String _name;
  private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

UTA

Extract Class (3)

```
class TelephoneNumber {
   String getAreaCode () {
      return _areaCode;
   }
   void setAreaCode (String arg) {
      _areaCode = arg;
   }
   String getNumber () {
      return _number;
   }
   void setNumber (String arg) {
      _number = arg;
   }

   private String _number;
   private String _areaCode;
}
```

UTA

Hide Delegate (1)

```
class Person {
   Department _department;

   public Department getDepartment () {
      return _department;
   }
   public void setDepartment (Department arg) {
      _department = arg;
   }
   …
}

class Department {
   private String _chargeCode;
   private Person _manager;

   public Department (Person manager) {
      _manager = manager;
   }
   public Person getManager () {
      return _manager;
   }
   …
}
```

UTA

Hide Delegate (2)

```
class Person {
   Department _department;

   public Department getDepartment () {
      return _department;
   }
   public void setDepartment (Department arg) {
      _department = arg;
   }
   public Person getManager () {
      return _department.getManager ();
   }
   …
}
```

```
manager = john.getDepartment().getManager()
```

⇩

```
manager = john.getManager()
```

UTA

Simplifying Conditional Expression (1)

- Decompose Conditional
  - Extract methods from the condition, then part and else part.
- Consolidate Conditional Expression
  - Combine multiple conditional tests with the same result
- Consolidate Duplicate Conditional Fragments
  - Move a code fragment that is in all branches of a conditional outside of the expression
- Remove Control Flag
  - Use break or return to remove a variable that is acting as a control flag

UTA

Simplifying Conditional Expression (2)

- Replace nested Conditional with Guard Clauses
  - Use guard clauses for special cases (or abnormal behavior)
- Replace Conditional with Polymorphism
  - If a conditional chooses different behavior based on the type of an object, replace it with polymorphism
- Introduce Null Object
  - Replace the null value with a null object to avoid repeated checks for a null value
- Introduce Assertion
  - Make assumptions explicit using assertions

UTA

Decompose Conditional

```
if (date.before(SUMMER_START) || (date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * summerRate;
```

⇩

```
if (notSummer(date))
    charge = winterCharge (quantity);
else charge = summerCharge (quantity);

private boolean notSummer (Date date) {
    return date.before (SUMMER_START) || date.after (SUMMER_END);
}
private double summerCharge (int quantity) {
    return quantity * _summerRate;
}
private double winterCharge (int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

UTA

Consolidate Duplicate Conditional Fragments

```
if (isSpecialDeal ()) {
   total = price * 0.95;
   send();
}
else {
   total = price * 0.98;
   send();
}
```

⇩

```
if (isSpecialDeal ()) {
   total = price * 0.95;
}
else {
   total = price * 0.98;
}
send ();
```

Remove Control Flag (1)

```
void checkSecurity (String[] people) {
   boolean found = false;
   for (int i = 0; i < people.length; i ++) {
       if (! found) {
          if (people[i].equals("Don")) {
              sendAlert ();
              found = true;
          }
      }
}
```

Remove Control Flag (2)

```
void checkSecurity (String[] people) {
   for (int i = 0; i < people.length; i ++) {
         if (people[i].equals("Don")) {
              sendAlert ();
              break;
          }
      }
}
```

UTA

Replace Nested Conditional with Guard Clauses

```
double getPayAmount () {
   double result;
   if (_isDead) result = deadAmount ();
   else
     if (_isSeparated) result = separatedAmount ();
     else {
        if (_isRetired) result = retiredAmount ();
        else result = normalPayAmount ();
     }
    return result;
}
```

⇩

```
double getPayAmount () {
   if (_isDead) return deadAmount ();
   if (_isSeparated) return separatedAmount ();
   if (_isRetired) return retiredAmount ();
   return normalPayAmount ()
}
```

UTA

Software Refactoring

- ❑ Introduction
- ❑ A Motivating Example
- ❑ Bad Smells in Code
- ❑ Catalog of Refactorings
- ❑ Summary

UTA

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.
- Bad smells help to decide when to refactor; the catalog of refactoring contains a common list of refactoring patterns.