8/10 - Good job. This was a reasonable copy from the book, but it really didn't explain or show the refactorings very well from your own perspective. The assignment was to try the code and I did see evidence of that beyond the typed in output in the end. A test case would have been more convincing.
The Form Template Method material was on page was on page 33 and refers to page 345 for the answer to the extra credit.

Anyway, good job.

David Pick
CM 2403
CSSE 375

Tests I ran before refactoring and their output:

```
    Customer tempCust = new Customer("David");
    Movie tempMovie = new Movie("LOTR", 1);
    Movie tempMovie2 = new Movie("Castle", 2);
    Movie tempMovie3 = new Movie("375", 0);
    Rental tempRentall1 = new Rental(tempMovie, 3);
    Rental tempRentall2 = new Rental(tempMovie2, 3);
    Rental tempRentall3 = new Rental(tempMovie3, 3);

    tempCust.addRental(tempRentall1);
    tempCust.addRental(tempRentall2);
    tempCust.addRental(tempRentall3);


    System.out.println(tempCust.statement());
```

Output:

```
Rental Record for David
LOTR 9.0
Castle 1.5
375 3.5
Amount owed is 14.0
You earned 4 frequent renter points
```

This must be the before picture on Customer? Don't know what happened to the formatting here.

```
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";
      while (rentals.hasMoreElements()) {
         double thisAmount = 0;
         Rental each = (Rental) rentals.nextElement();
```

```
            thisAmount = amountFor(each);

            // 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;

    }

    private int amountFor(Rental each) {
   int 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;
    }


}
```

Extract Method:
//In this step the book removed the large switch statement from the statement method.
//This refactoring makes the original statement method easier to read. It also makes the switch
//much easier to test, since now we can write a unit that that simply calls the function
//and checks to see if it works correctly

```
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";
       while (rentals.hasMoreElements()) {
          double thisAmount = 0;
          Rental each = (Rental) rentals.nextElement();

          thisAmount = each.getCharge();

          // 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;

   }

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

}
}
```

Good...

Move Method:

```
//This step moves the amountFor method that was created in the last step into the Rental
class.
//This is done because the rental class contains all the information needed to for the
amountFor
//method. This helps to maintain high cohesion and low coupling.
-----------------------------------------------------------------------------------------

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";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            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 ++;

            //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;

    }

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

```
}
}
```

Replace Temp with Query:
//In this step the book removes all calls to thisAmount and replaces them with the call
//each.getCharge(). This can be done because thisAmount is redundant and adds to the amount of
//code without reducing confusion or complexity. By elminating it, cleaner more readable code is
//createcd.

```
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() {
       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;
    }


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

}
}
```

Extract Method:
//In This step the getTotalCharge() method is created. It is created to so that the logic
//calculating the total cost of a rental is removed from the statement method. This allows
//both methods to be properly tested, and ensures that both methods work as intended.
//By creating the getTotalCharge method, the book has made the statement method more
//readable, understandable, and testable.

```
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() {
        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 int getTotalFrequentRenterPoints(){
```

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


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

}
}
```

Replace Temp with Query:
//This step the frequent renter points temporary variable. As was said before,
//this makes the code more readable and understandable.

```
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;
     }

     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;
    }

}
```

Move Method
This refactoring step adds another method to Rental, that was previously in Customer. This is done because Rental has the information necessary for the method and having it in Customer was creating higher coupling.

```
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;
    }

    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;
    }

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

}
```

Move Method
In this step the book moves the getCharge method and moves it from Rental to Movie, this was done to reduce coupling and make the Rental class more readable.

```
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;
    }

    double getCharge() {
      return _movie.getCharge(_daysRented);
    }

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

}
```

Final Form of the Rental Class

```
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;
    }

    double getCharge() {
      return _movie.getCharge(_daysRented);
    }

    int getFrequentRenterPoints() {
     return _movie.getFrequentRenterPoints(_daysRented);
    }

}
```

```
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;
   };
}
```

Original

```
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;
   }

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

}
```

Move Method

```
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 name, int priceCode) {
      _name = name;
      setPriceCode(priceCode);
    }
   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;


  public String getTitle (){
      return _title;
  }

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

}

 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;
  }
 }
```

Replace Type Code with State/Strategy:
This refactoring step replaced an if statement that was dependent on an integer in the class to determine the price of the movie, and created an abstract class called Price. This makes the code more readable and understandable. Since separate price objects will be created for different kinds of movies, it allows us to create different pricing strategies.

```
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 name, int priceCode) {
     _name = name;
     setPriceCode(priceCode);
   }
  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;


  public String getTitle (){
      return _title;
  }

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

   double getCharge(int daysRented) {
      return _price.getCharge(daysRented);
   }


}

 abstract class Price {
  abstract int getPriceCode();
```

```
  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.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
               result += (daysRented - 3) * 1.5;
            break;
      }
      return result;
   }

}
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;
 }
}
```

Replace Condition with Polymorphism
This step moves the conditional logic that had been in the movie class, into the abstract class for Price. This makes sure that all code determining the price of a movie is in the Price class. Making the code more understandable.

```
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 name, int priceCode) {
      _name = name;
```

```
    setPriceCode(priceCode);
   }
  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;


  public String getTitle (){
     return _title;
  }

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

   double getCharge(int daysRented) {
      return _price.getCharge(daysRented);
   }


}

 abstract class Price {
  abstract int getPriceCode();

  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.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
               result += (daysRented - 3) * 1.5;
            break;
       }
       return result;
   }

    int getFrequentRenterPoints(int daysRented){
        return 1;
    }



}

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

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

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

}
```

Final form of the Movie Class

The assignment also asked for an example of the Form Template Method pattern. Unfortunately there was no place in the code where I thought this could be applied. The pattern says that if you have to classes that are subclasses of the same object, and both have a method that preforms similar steps. You should create one method that can work for both objects and pull that object up into their super class.

There was a place with HTML parts ... in the book it says to go to page 345.

After each refactoring step I ran the same test code to ensure that no change to functionality had been made. After each step my test code outputted the following, which meant that I had successfully refactored the code.

Output:
Rental Record for David
LOTR 9.0
Castle 1.5
375 3.5
Amount owed is 14.0
You earned 4 frequent renter points