Foundations of Software Engineering

Refactoring

Fall 2017
Department of Computer Science Ben-Gurion University

Based on slides of:

Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Mira Balaban Department of Computer Science Ben-Gurion university
F. Tip. IBM T J Watson Research Center.
Refactoring: Improving the Design of Existing Code

Martin Fowler  
Kent Beck  
John Brant  
William Opdyke Don Roberts  
Publisher: Addison Wesley  
FirstEdition June 11, 1999  
ISBN: 0-201-485672, 464 pages

“Computer science is the discipline that believes all problems can be solved with one more level of indirection”

Dennis DeBruler (from Fowler’s Refactoring book)
What is refactoring?

- refactoring is the process of applying transformations (refactorings) to a program, with the goal of improving its design.

- goals:
  - keep program readable, understandable, and maintainable.
  - by eliminating small problems soon, you can avoid big trouble later.

- two key features of refactorings:
  - behavior-preserving: make sure the program works after each step.
  - typically small steps
Why refactor?

- why does refactoring become necessary?
  - requirements have changed, and a different design is needed.
  - design needs to be made more flexible (so new features can be added).
  - sloppiness by programmers (e.g., cut-and-paste programming when introduction of a new method).
  - programmers usually don’t come up with the ultimate design right away.
  - refactoring often has the effect of making a design more flexible.
  - design patterns are often a target for refactoring.
History

• Refactoring is something good programmers have always done.
• Especially important in the context of object-oriented languages.
  • Perhaps because object-oriented features are well-suited to make designs flexible and reusable.
  • But refactoring is really not specific to OO.
• Opdyke’s PhD thesis (1992) describes his research on refactoring tools for Smalltalk.
  • Various other students of Ralph Johnson have worked on refactoring tools, mostly for Smalltalk.
• Refactoring is becoming very popular due to “lightweight” development methodologies such as extreme programming that advocate continuous refactoring.
Preserving program behavior

- How to ensure that the program does the same thing before and after applying a refactoring?

  - **Testing**: write tests that exercise the parts of the program affected by the refactoring.
    - In general, no guarantees.

  - **Program analysis**: Perform a static analysis of the program using techniques similar to those used in compilers.
    - Difficult to implement; analysis may be imprecise and say that a refactoring cannot be applied safely.
    - Some refactoring support is incorporated in Eclipse and IntelliJ.
Fowler’s book:


- Refactoring (noun):
  a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

- Refactor (verb):
  to restructure software by applying a series of refactorings.
Fowler’s book:

- Provides a catalogue of refactorings, similar to the catalogue of design patterns in the GoF book.
- Catalogues “bad smells” --- indications that refactoring may be needed.
- Explains when to apply refactorings:
  - UML diagrams to illustrate the situation before and after.
- Examples of code before and after each refactoring.
  - Small examples that are representative of larger systems.
- Many of Fowler’s refactorings are the inverse of another refactoring.
  - Often there is not a unique “best” solution.
  - Discussion of the tradeoffs.
“Bad smells”: An indication that the design may not be optimal

• Just a sample:
  • Duplicated code (cut & paste programming).
  • Long method.
  • Large class.
  • Long parameter list.
  • Primitive obsession.
  • Switch statements.

• Some of the more controversial ones:
  • Speculative generality.
  • Comments.

Example: Refactorings applied

• Straight from the book:

  • “A program to calculate and print a statement of a customer’s charges at a video store”. There are three types of movies: Regular, NewRelease, Children
  • Rates —
    • Regular: 2.00 for 2 days; late 1.50/late day
    • Children: 1.50 for 3 days; late 1.50/late day
    • NewRelease: 3.00/day

• Price depends on how long the movie is rented and the category of the movie.

• Also compute frequent renter points.
Example: Refactorings applied

- Class diagram of the starting point classes.

  - Movie: a simple data class
  - Rental: The rental class represents a customer renting a movie.
  - Customer: represents the customer of the store
Example: Movie class

```java
public class Movie {
    public static final int REGULARS = 0;
    public static final int NEW_RELEASE = 1;
    public static final int CHILDREN = 2;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public String getTitle() {
        return _title;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }
}
```
Example: Rental Class

```java
public 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;
    }
}
```
public 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 class Customer
...

    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;
            }

            totalAmount += thisAmount;
            frequentRenterPoints += each.getFrequentRenterPoints();
        }

        result += String.format("Total Amount: %.2f\n", totalAmount);
        result += String.format("Frequent Renter Points: %d\n", frequentRenterPoints);
        return result;
    }

Example: Customer Class (2)
Example: Customer Class (3)

```java
public class Customer {
    public String statement() {
        ...
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount+=(each.getDaysRented()-3) * 1.5;
            break;
        }
        // end switch
        // 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++;
    }
```
Example: Customer Class (4)

```java
public class Customer
    public String statement()
        ...
        //show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    } // end while

    // add footer lines
    result += "Amount owed is +
            String.valueOf(totalAmount) + "\n";
    result += "You earned +
            String.valueOf(frequentRenterPoints) + "frequent renter points\n";
    return result;
```
public class Customer
...

    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;
            
            return result + each + “\n”;
        }
    }
public class Customer
...

    public String statement(){
        // initializations
        while (rentals.hasMoreElements()) {
            // initializations
            // determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR: ...
                case Movie.NEW_RELEASE: ...
                case Movie.CHILDRENS: ...
            } // end switch

            // add frequent renter points
            // add bonus for a two day new release rental
            // show figures for this rental: result += ...; totalAmount += ...;
        } // end while
        // add footer lines: result += ...;
        return result;
    }
Example: Customer.statement()

- Interaction diagram for Customer.statement():
  - Customer does everything!
Changing requirements: A trigger for refactoring:

- Add an htmlStatment method which returns a customer statement string containing html tags
  - requires code duplication.
  
  ...and

- There will be some changes to the way movies are classified.
  ...affecting frequent renter points and charging.
  ...developer estimation: they will change it again within six months

- NOTE: The code works well!

When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.
Refactoring prerequisite

- Write a test suite – recall the TDD development approach!
  - Make sure: All tests are passed.

- Refactoring should not affect the outcome of tests.
- The test suite must exercise the published interface of the classes.

- Refactoring should not affect the *published* interface.
- So, avoid publishing interfaces too early.
Refactoring: step 1 – extract method

- Customer.statement() is too long.
- Should be decomposed into smaller pieces.
- Find a logical part and use the extract method refactoring:
  - The switch statement.
  - Handle local variables and parameters:
    - `each` is not modified by the code - can be passed in as a parameter
    - `thisAmount` is modified – if unique, can be returned as the result
    - Local to extracted code – declare in target method.
Refactoring: step 1a – the extracted code

public String statement() {
    ...
    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;
        }
    ...
}
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n";
    return result;
}
Refactoring: step 1c – the extracted method

public class Customer{
  ...
  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;
  }
}
Test: step 1

Oops,  (double) -> (int) bug!
Java compiler won’t catch it! Only a good test case.

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

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

Variable names not helpful

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

```java
public 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;
}
```

Test – all tests are passed!

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
Refactoring: step 3 – Move method

Moving amount computation (does not use info from Customer only from Rental)

```java
class Customer {

public 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;
}
```
Refactoring: step 3 – Move method

- Steps:
  - Copy code to Rental.
  - Adjust the copied code:
    - Remove parameter.
    - Rename (amountFor → getCharge).
  - Compile and test.
  - Change references to the old method.
  - Compile and test.
  - Remove the old method.
Refactoring: step 3a – the new method is Rental.getCharge()

class Rental ...
public 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 ;
}
Refactoring: step 3a – the new method is Rental.getCharge()

class Customer …

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

Compile and test!

• Note:
We could leave the old method to delegate to the new method. This is useful if it is a public method and we don't want to change the interface of the other class.
Step 3b – change references to the old method:

```java
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n";
    return result;
}
```
Step 3b – change references to the old method:

```java
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n";
    return result;
}
```
Step 3b – change references to the old method:

```java
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n";
    return result;
}
```
Refactoring: After step 3

- State of classes after moving the charge method.
- Customer.amountFor() is deleted.
Refactoring: Step 4 – replace temp with query

class Customer …

   // thisAmount is redundant.

   public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
      Enumeration rentals = _rental.elements();
      String result = “Rental Record for “ + getName() + “\n”;
      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 ++;
         // 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\n”;  
   return result;
}
Refactoring: step 5 – extract method and move method

- Back to Customer.statement().
- Extract frequent renter per movie points.
  - Handle local variables and parameters:
    - Used in source method – pass as parameters (each).
    - Modified – if unique, return as the result (frequentRenterPoints).
      But here – target does not rely on value of frequentRenterPoints as input
- Move the extracted method to Rental.
Refactoring: step 5 – the extracted code

class Customer ...

public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.elements();
    String result = "Rental Record for " + getName() + "\n";
    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++;
        // 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\n";
    return result;
}
class Customer {

    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rental.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\n”;
        return result;
    }
}
Refactoring: step 5c – the extracted and moved method

class Rental …

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

Compile and test!
Summary of refactoring step 5

- Class diagram before extraction and movement of the frequent renter points calculation

- Interaction diagram before extraction and movement of the frequent renter points calculation
Summary of refactoring step 5

- Class diagram after extraction and movement of the frequent renter points calculation

- Interaction diagram after extraction and movement of the frequent renter points calculation
Refactoring: step 6 – replace temp with query

class Customer …

public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n”;
    return result;
}
Refactoring: step 6a – replace temp with query

class Customer ...

public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.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\n”;
    return result;
}
Refactoring: step 6b – the totalCharge query

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

NOTE: This isn't the simplest case of **Replace Temp with Query** totalAmount was assigned to within the loop, so we had to copy the loop into the query method.
Refactoring: step 6 – replace temp with query

class Customer …

public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rental.elements();
    String result = “Rental Record for “ + getName() + “
    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 “\n”;
    return result;
}
Refactoring: step 6c – replace temp with query

class Customer …

public String statement() {
    Enumeration rentals = _rental.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\n”;    
    return result;
}
class Customer ...

    private double getTotalFrequentRenterPoints() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getFrequentRenterPoints();
        }
        return result;
    }
Summary of refactoring step 6

- Class diagram before extraction of the totals

- Interaction diagram before extraction of the totals
Summary of refactoring step 6

- Class diagram after extraction of the totals

- Interaction diagram after extraction of the totals
Comments on refactoring step 6

- Most refactoring reduce code size, but this is not necessarily the case.
- The point is to make code easier to modify and more readable.

- Performance gets a hit by running the same loop three times, or maybe not?
  - Profile the program and find the answer.

- Functionality can be extended, e.g., adding Customer.htmlStatement() without duplicating the computation of rental charges, and frequent renter points.
"I am now at the point where I take off my refactoring hat and put on my adding function hat."

- write htmlStatement as follows and add appropriate tests:

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

• Getting ready to change the classification of the movies in the store.

• Perhaps new classification, perhaps modification to existing.

• Charging and frequent renting will be affected.

➔ improve the charge and frequent renter point methods.
➔ Replace conditional logic on Price Code with polymorphism
Refactoring: step 7 – move method

- It is a bad idea to do a switch based on an attribute of another object.
- Move `getCharge` – switch on an attribute of another object.

```java
class Rental ...
public 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;
}
```
Refactoring: step 7 – move method – where to and why?

- *Rental.getcharge()* switches on an attribute of *rental._movie* that varies with the movie type.

- *Rental.getcharge()* uses also data from *Rental (_daysRented)*.
  - If in *Movie*  *Movie.getCharge()* uses data from *Rental*.

- **Preferred since** types change frequently.
  - Type information generally tends to be more volatile
  - Changing *Movie* types  least possible dependencies.

- Note: If a rental object is passed to a Movie  increase coupling.
Refactoring: step 7a – The new method

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

The method effectively uses two pieces of data, the length of the rental and the type of the movie
Refactoring: step 7b – The old method

class Rental ...
public double getCharge() {
    return _movie.getCharge(_daysRented);
}

Refactoring: step 8 -
moving method – move frequent renter point calculation from Renter to Movie

- Move getFrequentRenterPoints() since it varies with the movie type.

```java
class Rental {
    public int getFrequentRenterPoints() {
        if ((getMovie().getPriceCode() == Movie.NEW_RELEASE)
            && getDaysRented() > 1)
            return 2;
        else
            return 1;
    }
```
Refactoring: step 8 – move method – move frequent renter point calculation from Renter to Movie

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

class Rental ...
public int getFrequentRenterPoints() {
    return _movie.getFrequentRenterPoints(_daysRented);
}
Refactoring: after step 8

- Class diagram before moving methods to movie

- Class diagram after moving methods to movie
Refactoring:
Insert inheritance by subclassing

- Insert subclasses.
- Replace *switch* by polymorphism.
- We have several types of movie that have different ways of answering the same question.
- We can have three subclasses of movie
  - each can have its own version of charge

→ The subclasses are Movies’ states.
→ Problem: A movie can change its class during its lifetime!
Refactoring: Use the State pattern.

- Find out about *Movie* states:
  depend on the price (the \_priceCode attribute of *Movie*).
- Insert a *Price* abstract class: Represents a movie’s state (e.g., new release).
- Subclass *Price*.
- Strategy is also possible.

price class represent an algorithm for calculating the price ➔ Strategy
Price class represent a state of the movie ➔ State
Refactoring: next steps

- **Step 9**: Move the type code behavior into the *State* pattern (*Replace Type Code with State/Strategy*)
  - Move `_priceCode` behavior to the *state* classes.
  - Modify the *state* accessors — connect the *Context* (*Movie*) with an *Actual State* (*NewReleasePrice, ChildrenPrice, RegularPrice*).

- **Step 10**: Move the *Movie.getCharge()* state dependent method to the *Price* class (*Move Method*).

- **Step 11**: Refactor the *Price.getCharge()* method — Eliminate the *switch* statement (*Replace Conditional with Polymorphism*).

- **Step 12**: Move the *Movie.getFrequentRenterPoints()* state dependent method to the *Price* class (*Move Method*).

- **Step 13**: Override the *Price.getCharge()* method.
Refactoring: step 9 – Replace Type Code with State/Strategy

- **Step 9a:** Encapsulate the type code (the `_priceCode` attribute), so to ensure no direct references.

- Use the *Self Encapsulate Field* refactoring:

  ```java
  class Movie ...
  public Movie(String name, int priceCode) {
    _name = name;
    _priceCode = priceCode;
  }
  ```

  After refactoring (there was a single direct reference):

  ```java
  class Movie ...
  public Movie(String name, int priceCode) {
    _name = name;
    setPriceCode(priceCode);
  }
  ```

**Compile and test!**
Refactoring: step 9b – Add the new classes

- Put the type code behavior in the new classes – the \textit{price code}:

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

class ChildrenPrice extends Price {
    int getPriceCode() {
        return MOVIE.CHILDREN;
    }
}

class NewReleasePrice extends Price {
    int getPriceCode() {
        return MOVIE.NEW_RELEASE;
    }
}

class RegularPrice extends Price {
    int getPriceCode() {
        return MOVIE.REGULAR;
    }
}
```
Refactoring: step 9c – change accessing to the moved type code:

- change Movie's accessors for the type code (_priceCode) to use the new classes:
- Accessors before change:

```java
class Movie ...

public int getPriceCode() {
    return _priceCode;
}
public void setPriceCode(int arg) {
    _priceCode = arg;
}
private int _priceCode;
```
Refactoring: step 9c – modified accessors

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

Compile and test!
Refactoring: step 10 – Move Method – from Movie.getCharge()

class Movie ...
public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case REGULAR:
            result += 2;
            if (daysRented > 2)
                result += (daysRented - 2) * 1.5;
            break;
        case NEW_RELEASE:
            result += daysRented * 3;
            break;
        case CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
                result += (daysRented - 3) * 1.5;
            break;
    }
    return result ;
}
Refactoring: step 10a – Move Method – to Price.getcharge()

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.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
                result += (daysRented - 3) * 1.5;
            break;
    }
    return result;
}
Refactoring:
step 10b – adjust the source method

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

Compile and test!
Refactoring: step 11 – Replace Conditional with polymorphism – in Price.getcharge()

```java
class RegularPrice ...

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

class NewReleasePrice ...

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

class ChildrenPrice ...

double getCharge(int daysRented) {
    double result = 1.5;
    if (daysRented > 3)
        result += (daysRented-3) * 1.5;
    return result;
}

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

// Replace the conditional “legs” one at a time.
// Override Price.getCharge()
// Compile and test!
Refactoring: step 12 – Move Method – from Movie.getFrequentRenterPoints() ...

class Movie...

    int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) &&
            daysRented > 1)
            return 2;
        else
            return 1;
    }
Refactoring: step 12 – Move Method – to Price.getFrequentRenterPoints()

```java
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;
    }
}
Refactoring: step 13 – Override the Price.getFrequentRenterPoints() method

- Extra frequent renter points are given to New releases rentals only.

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

class NewReleasePrice..
    int getFrequentRenterPoints(int daysRented) {
        return (daysRented > 1) ? 2:1;
    }
Refactoring: Object interaction in the final Customer

statement()
Refactoring: The final class diagram
Refactoring example – Evaluation

• Insertion of the *State* pattern required much refactoring.

• Advantage: Price code dependent information and changes do not affect the rest of the system.
  • Changing the rules for charging and frequent renter points calculation is independent from the rest of the system.
  • Changing the classification of movies is easy.

• Mode of writing -- as in TDD: test, small change, test…

• Replaces the need for debugging.
Refactorings used in the Video Store example

- Extract method.
- Rename variable.
- Move method.
- Replace temp with query.
- Replace type code with state/strategy.
- Encapsulate field.
- Inline temp (as part of Replace temp with query).
Refactoring for Visitor (1)— example following Mens & Tourwe, 2004

• **Document class hierarchy and helper classes:**
  
  • *Document*, with *print()*, *preview()*.
  
  • *Document* subclasses:
    • *ASCIIDoc* with *print{X}*, *preview{A}*. 
    • *PSDoc* with *print{Y}*, *preview{B}*. 
    • *PDFDoc* with *print{Z}*, *preview{C}*. 
  
  • *Document* helper classes:
    • *PreViewer* with *preview(Document)*. 
    • *Printer* with *print(Document)*. 

• **Problems:**
  
  • Document functionalities are spread around. 
  • Adding Document functionalities (e.g., text search or a spell checker) is difficult (we need to change every subclass of Document and we need to define the appropriate helper classes). 
  • Document class has many associations. 
  • Similarity among Document helper classes is lost (although their roles are similar)
Refactoring for Visitor (1a)– example following Mens & Tourwe, 2004

Fig. 1. *Document* class hierarchy and helper classes
Refactoring for Visitor (1b) – example following Mens & Tourwe, 2004

Fig. 2. Refactored design model for the Document class hierarchy
Refactoring for Visitor (2) – example following Mens & Tourwe, 2004

- **Document class hierarchy and Visitor classes:**
  - *Document* with:
    - `print() {this.accept(new Printer()) }`
    - `preview() {this.accept(new Previewer()) }`
    - `Accept(Visitor v)`
  - *Document subclasses*:
    - `ASCIIDoc` with `Accept(Visitor v) {v.visitASCII(this)}`.
    - `PSDoc` with `Accept(Visitor v) {v.visitPS(this)}`.
    - `PDFDoc` with `Accept(Visitor v) {v.visitPDF(this)}`.
  - *Visitor with:* `visitASCII(ASCIIDoc d)`, `visitPS(PSDoc d)`, `VisitPDF(PDFDoc d)`.
  - *Visitor subclasses*:
    - `Printer` with `visitASCII(ASCIIDoc d) {X'}`, `visitPS(PSDoc d) {Y'}`, `VisitPDF(PDFDoc d) {Z'}`.
    - `Previewer` with `visitASCII(ASCIIDoc d) {A'}`, `visitPS(PSDoc d) {B'}`, `VisitPDF(PDFDoc d) {C'}`.
Refactoring for Visitor (3)– example following Mens & Tourwe, 2004

Primitive refactorings involved in the insertion of the Visitor design pattern:

1. **RenameMethod**: 3 `print` methods in `Document` subclasses are renamed into `visitASCII`, `visitPS`, `visitPDF` methods.
2. **MoveMethod**: 3 visit methods moved to the `Printer` class.
3. **RenameMethod**: 3 `preview` methods in `Document` subclasses are renamed into `visitASCII`, `visitPS`, `visitPDF` methods.
4. **MoveMethod**: 3 visit* methods moved to the `PreViewer` class.
5. **AddClass**: An abstract superclass `Visitor` for `Printer` and `PreViewe`r is added.
6. **AddMethod**: 3 visit* methods added to the `Visitor` class.
7. **AddMethod**: Add `accept`, `print`, `preview` to `Document` subclasses.
8. **PullUpMethod**: Pull the `print` and `preview` methods from `Document` subclasses to `Document`. 
Refactoring for Visitor (3a) – example following Mens & Tourwe, 2004

- **Composite** refactoring for renaming and moving print methods from the Document subclasses to the Printer class (primitive refactorings 1 and 2)

![Diagram showing composite refactoring](image)

Fig. 3. Composite refactoring for renaming and moving *print* methods from the *Document* subclasses to the *Printer* class
Some kinds of Refactorings

• **Primitive refactorings**: e.g., RenameMethod, MoveMethod, AddClass, AddMethod, PullUpMethod, ExtractMethod.

• **Composite refactorings**: e.g., Extract&MoveMethod, Extract&PullUpMethod.

• **Refactoring for design patterns**: e.g., MoveMethodsToVisitor, Replace type code with State/Strategy.

• **Big refactorings**: e.g., Convert procedural design to objects, Extract hierarchy.
Refactoring activities

- Identify **where** to apply.
- **Bad smells.**
- Determine **which** refactoring should be applied.

- **Guarantee** that the applied refactoring **preserves behavior.**

- **Apply** the refactoring.

- **Assess** the effect of the refactoring on the quality of the software
  - Performance, complexity, understandability, maintainability, productivity, cost, effort.

- Maintain consistency between the refactored program code and other software artifacts.
Refactoring Principles

- **Why** do we refactor?
  - To improve the design of software
  - To make software easier to understand
  - To help you find bugs
  - To make you program faster

- **When** should we refactor?
  - Refactor when you add functionality
  - Refactor when you need to fix a bug
  - Refactor as you do code reviews
  - Refactor when the code starts to smell.

- **What** about performance?
  - Worry about performance only when you have identified a performance problem
What is the difference between

- Refactoring
- Debugging
- Code restructuring
- Design patterns
Bad Smells in Code

• If it stinks, change it.
  ---Grandma Beck on child rearing

• Duplicated Code
  
  • If the same code structure is repeated
  • Extract Method - gather duplicated code
    • Simplest – duplication in the same class.
  • Pull Up Method - move to a common parent
  • In sibling classes. Extract method + Pull Up Method.
  • Form Template Method - gather similar parts, leaving holes.
  • Similar but not equal code in sibling classes.
  • Substitute Algorithm - choose the clearer algorithm
  • Extract class - create a new class with the duplicated code. For duplication in unrelated classes.
Bad Smells in Code

- Long Method

- If the body of a method is over a page (choose your page size)

- Extract Method - extract related behavior. The need for comments is a good heuristic.
- Replace Temp with Query - remove temporaries when they obscure meaning.
  - Might enable extract method.
- Introduce Parameter Object / Preserve Whole Object - slim down parameter lists by making them into objects.
  - Extract Method might lead to long parameter lists.
- Replace Method with Method Object – If still too many parameters. Heavy machinery.
- Decompose Conditionals - conditional and loops can be moved to their own methods
Bad Smells in Code

- **Large Class**
  
  - If a class is doing too much:
  - has too many variables or too many methods

- Extract Class - to bundle variables or methods.
- Extract Subclass – A class has features that are used only by some instances.
- Extract interface – determine how clients use the class. Provide ideas on breaking the class.
Bad Smells in Code

- Long Parameter List

  - A method does not need many parameters, only enough to be able to retrieve what it needs.
  - Long parameter lists are hard to understand and maintain.
  - The clue – pass objects: Use objects for packing data.
  - Penalty – might increase dependency among objects.

- Replace Parameter with Method - An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
  - Remove the parameter and let the receiver invoke the method.

- Preserve Whole Object – replace parameters that result from an object by the object itself.

- Introduce Parameter Object - turn several parameters into an object.
Bad Smells in Code

• **Divergent Change**

• If you find yourself repeatedly changing the same class for different requirement variations – then there is probably something wrong with it.

• A class should react to a single kind of variation – cohesion principle.

• Extract Class - group functionality commonly changed into a class
Bad Smells in Code

- **Shotgun Surgery**

  - If you find yourself making a lot of small changes for each desired change.
  - Small changes are hard to maintain.
  - **Opposite** of divergent change.
    - divergent change = one class, many changes
    - Shotgun Surgery = one change, many classes
  - Ideal: common changes ↔ classes is a 1:1 relationships.

- **Move Method/Field** - pull all the changes into a single class (existing or new).
- **Inline Class** - group a bunch of behaviors together in an existing class (might imply divergent change).
Bad Smells in Code

- Feature Envy

  - If a method seems more interested in a class other than the class it actually is in – move it to where it belongs.
  - Strategy and Visitor break this rule – separate behavior from the data it works on. Answer the Divergent Change smell.

- Move Method - move the method to the desired class.
- Extract Method + Move Method - if only part of the method shows the symptoms.
- Or, if the method uses data from several classes.
Bad Smells in Code

- **Data Clumps**
  - Data items that are frequently together in method signatures and classes belong to a class of their own.
  - A test for a Data Clump: Delete one value and see if the others still make sense.
  - Extract Class - turn related fields into a class.
  - Introduce Parameter Object / Preserve Whole Object - for reducing method signatures.
  - Look for Feature Envy – Move Method.
Bad Smells in Code

• **Primitive Obsession**
  - **Primitive types inhibit change.**

• **Replace Data Value with Object** - on individual data values.

• If a primitive value is a type code:
  - Replace type Code with Class — The value does not affect behavior.
  - Conditionals on the type code —
    - Replace Type Code with Subclasses.
    - Replace Type Code with State/Strategy.

• **Extract Class** — a set of inter-related value fields.

• **Introduce Parameter Object** - for method signatures.
Bad Smells in Code

- **Switch Statements**

  - Switch statements lead to Code Duplication and inhibit change.
  - **Object-Oriented switch = Polymorphism.**

- If the switch is on a type code:
  - Extract method - to extract the switch.
  - Move method - to get the method where polymorphism applies.
  - Replace Type Code with State/Strategy / Replace Type Code with Subclasses - set up inheritance
  - Replace Conditional with Polymorphism - get rid of the switch.

- Few cases that affect a single method; Cases are stable- Polymorphism is overkill
  try:
  - Replace Parameter with Explicit Methods – if the switch value is a method parameter.
  - Introduce Null Object – If there is a conditional case comparing with null.
Bad Smells in Code

- Lazy Class
  - If a class (e.g. after refactoring) does not do much, eliminate it.

- Collapse Hierarchy- for subclasses.

- Inline Class - Move all its features into another class and delete it.
Bad Smells in Code

• Speculative Generality

  • If a class has features that are only used in test cases, remove them (and the test case)...

  • Think TDD!

• Collapse Hierarchy - for useless abstract classes.

• Inline Class - for useless delegation.

• Remove Parameter – methods with unused parameters.

• Rename Method - methods with odd abstract names should be brought down to earth.
Bad Smells in Code

- **Middle Man**
  - An intermediary object is used too often to get at encapsulated values.
  - Too many methods are just delegating behavior.

- Remove Middle Man - to talk directly to the target.
- Inline Method – inline the delegating methods in their clients – if only few delegating methods.
- Replace Delegation with Inheritance - turn the middle man into a subclass of the target object.
  - Only if all methods of the target class are used by the Middle Man.
Bad Smells in Code

- **Refused Bequest**
  - A subclass refuses or does not need most of its heritage.
  - The hierarchy is wrong.

- **Push Down Method / Push Down Field** – create a sibling class. Push all unused methods to the sibling → parent holds only the common structure and functionality.

- **Replace Inheritance with Delegation** – get rid of wrong hierarchy.
Bad Smells in Code

- Comments
  
  - Comments are often a sign of unclear code… consider refactoring
  
  - Extract Method.
  - Rename Method.
  - Introduce Assertion.