7: refactoring

joão m fernandes

what is refactoring?

- refactoring is a disciplined way to clean up code that minimizes the chances of introducing bugs
- when you refactor you are improving the design of the code after it has been written
- refactoring changes a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure
- with refactoring you can take a bad, even chaotic, design, and rework it into well-designed code
- refactoring is a series of small steps, each of which changes the program's internal structure without changing its external behavior

reasons to refactoring (1)

- code is duplicated
- a routine is too long
- a loop is too long or too deeply nested
- a class has poor cohesion
- a class interface does not provide a consistent level of abstraction
- a parameter list has too many parameters
- changes within a class tend to be compartmentalized
- changes require parallel modifications to multiple classes

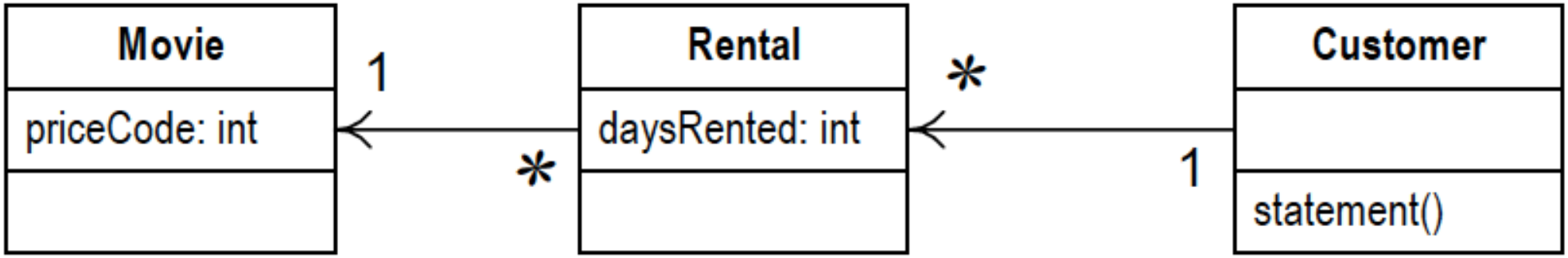
reasons to refactoring (2)

- inheritance hierarchies have to be modified in parallel
- case statements have to be modified in parallel
- related data items that are used together are not organized into classes
- a routine uses more features of another class than of its own class
- a primitive data type is overloaded
- a class doesn’t do very much
- a chain of routines passes tramp data
- a middleman object isn’t doing anything

reasons to refactoring (3)

- one class is overly intimate with another
- a routine has a poor name
- data members are public
- a subclass uses only a small percentage of its parents' routines
- comments are used to explain difficult code
- global variables are used
- a routine uses setup code before a routine call or takedown code after a routine call
- a program contains code that seems like it might be needed someday

example



- the program calculates and prints a statement of a customer's charges at a video store
- the program is told which movies a customer rented and for how long
- it calculates the charges, which depend on how long the movie is rented, and identifies the type movie
- 3 kinds of movies: regular, children's, new releases
- the statement also computes frequent renter points

```
public class Movie {
   public static final int REGULAR = 0;
   public static final int NEW_RELEASE = 1;
   public static final int CHILDRENS = 2;
   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;
   };
}
```

```
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;
  }
}
```

```
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;
  };
```

```
1  public class Customer {
2    public String statement() {
3      double totalAmount = 0;
4      int frequentRenterPoints = 0;
5      Enumeration rentals = _rentals.elements();
6      String result = "Rental Record for " + getName() + "\n";
7      while (rentals.hasMoreElements()) {
8        double thisAmount = 0;
9        Rental each = (Rental) rentals.nextElement();
10       //determine amounts for each line
11       switch (each.getMovie().getPriceCode()) {
12         case Movie.REGULAR:
13           thisAmount += 2;
14           if (each.getDaysRented() > 2)
15             thisAmount += (each.getDaysRented() - 2) * 1.5;
16           break;
17         case Movie.NEW_RELEASE:
18           thisAmount += each.getDaysRented() * 3;
19           break;
20         case Movie.CHILDRENS:
21           thisAmount += 1.5;
22           if (each.getDaysRented() > 3)
23             thisAmount += (each.getDaysRented() - 3) * 1.5;
24           break;
25       }
26       // add frequent renter points
27       frequentRenterPoints ++;
28       // add bonus for a two day new release rental
29       if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
30         each.getDaysRented() > 1) frequentRenterPoints ++;
31       //show figures for this rental
32       result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
33       totalAmount += thisAmount;
34     }
35     //add footer lines
36     result += "Amount owed is "+String.valueOf(totalAmount)+"\n";
37     result += "You earned " + String.valueOf(frequentRenterPoints)+
38       " frequent renter points";
39     return result;
40   }
41 }
```

example

- this program is not well designed
- many of the things that the **statement()** routine does should really be done by the other classes
- a poorly designed system is hard to change
- users would like a statement printed in HTML
- it is impossible to reuse any of the behavior of the current statement method for an HTML statement
- when the charging rules change, you have to fix both **statement()** and **htmlStatement()** and ensure the fixes are consistent

refactoring tips

when you find you have to add a feature to a program, and the program's code is not conveniently structured to add the feature, first refactor the program to make it easy to add the feature, then add the feature

before you start refactoring, check that you have a solid suite of self-checking tests

example

- the first refactorings show how to split up the long method and move the pieces to other classes
- the aim is to make it easier to write an HTML statement method with less duplication of code

extract method

if you have a code fragment that can be grouped together, turn the fragment into a method whose name explains the purpose of the method

```
void printOwing() {
  printBanner();
  System.out.println("name: " + _name);
  System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}
void printDetails(double outstanding) {
  System.out.println("name: " + _name);
  System.out.println("amount: " + outstanding);
}
```

example before

```
public class Movie {
  . . .

  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.CHILDRENS:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
      }
      // more code
```

example after

```
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;
}
```

example after

```
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;
}
```

change names of variables

- good code should communicate what it is doing clearly
- variable names are a key to clear code
- always change the names of things to improve clarity
- with good find-and-replace tools, it is usually easy
- strong typing and testing will highlight anything you miss

anyone can write code that a computer can understand
good programmers write code that humans can understand

example before

```
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;
}
```

example

after

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

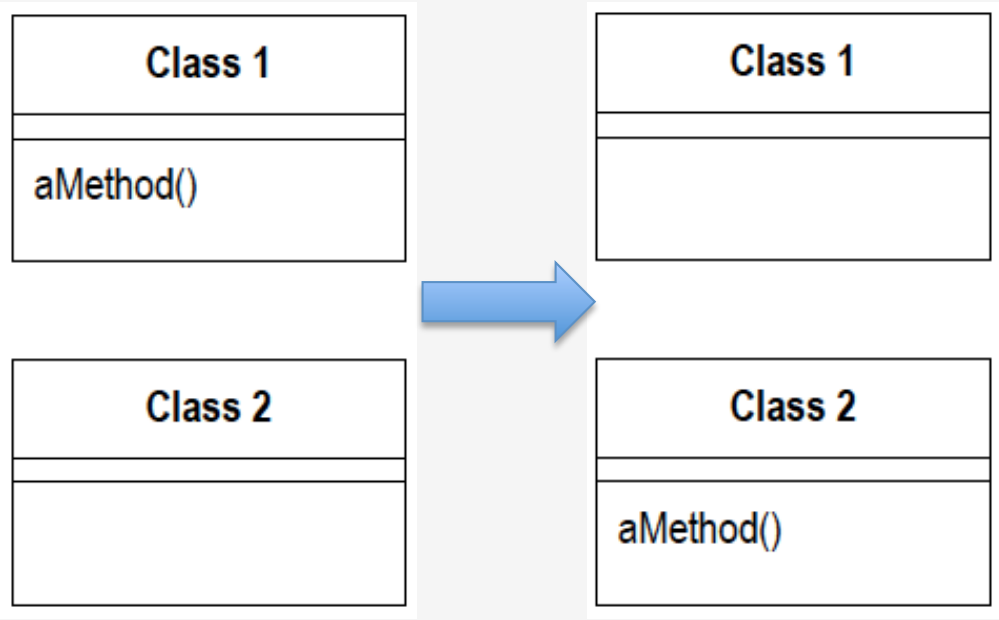
refactoring tip

refactoring changes the programs in small steps

if you make a mistake, it is easy to find the bug

move method

- a method is using or used by more features of another class than the class it is defined on
- create a new method with a similar body in the class it uses most
- either turn the old method into a simple delegation, or remove it altogether



- **amountFor()** uses information from the rental but not from the customer
- it is on the wrong object
- the method should be moved to rental

example before

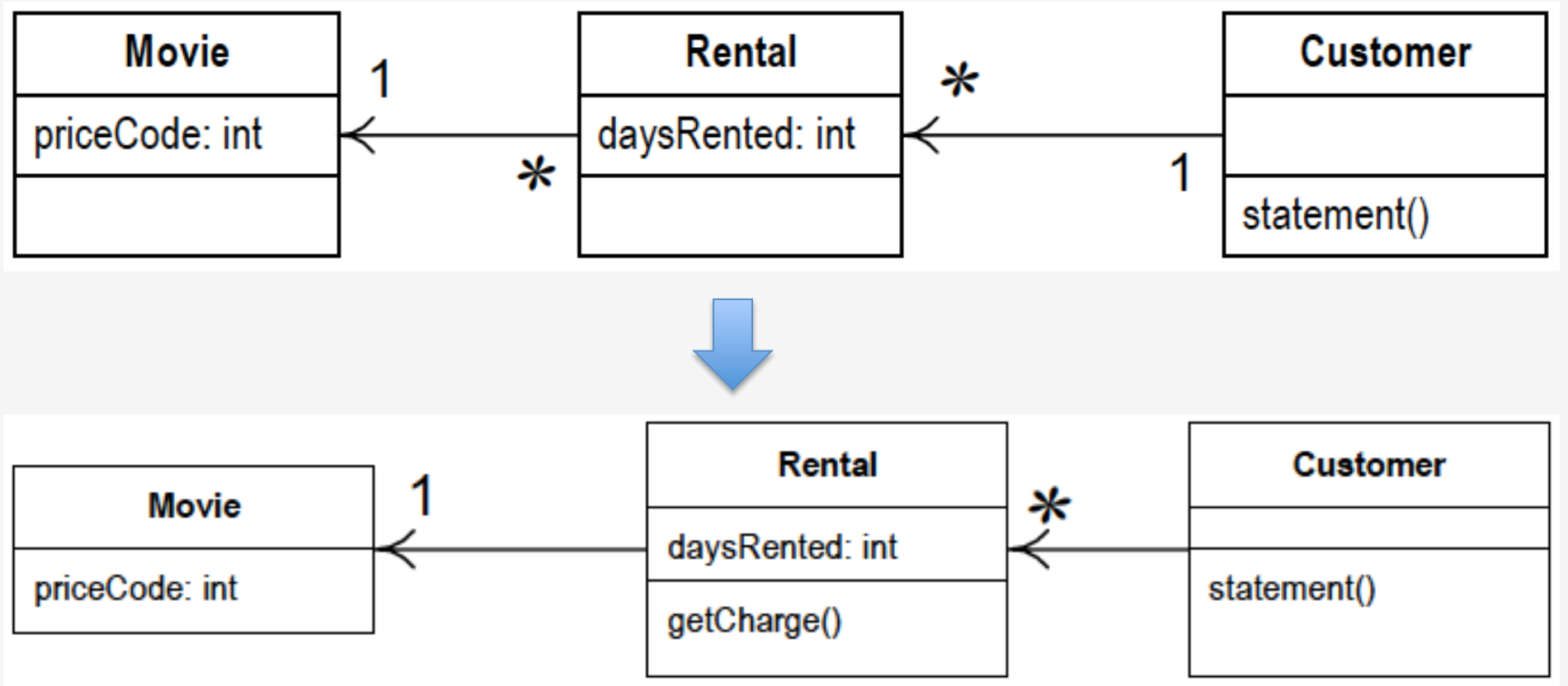
```
class Customer {
  . . .
  private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (aRental.getDaysRented() > 2)
          result += (aRental.getDaysRented() - 2) * 1.5;
        break;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

example
after

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

example after

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



example before

```
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();
      thisAmount = amountFor(each);
      // add frequent renter points
```

```
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();
      thisAmount = each.getCharge();
      // add frequent renter points
```

replace temp with query

- the attribute **thisAmount** is now redundant
- it is set to the result of **each.getCharge()** and not changed afterward
- we can eliminate **thisAmount**
- try to get rid of temporary variables such as this
- temps are often a problem since they cause a lot of parameters to be passed around when they don't have to be
- one can easily lose track of what they are there for
- there is a performance price to pay
- in this example, the charge is now calculated twice

example after

```
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 ++;
    // 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;
}
```

extract method: example before

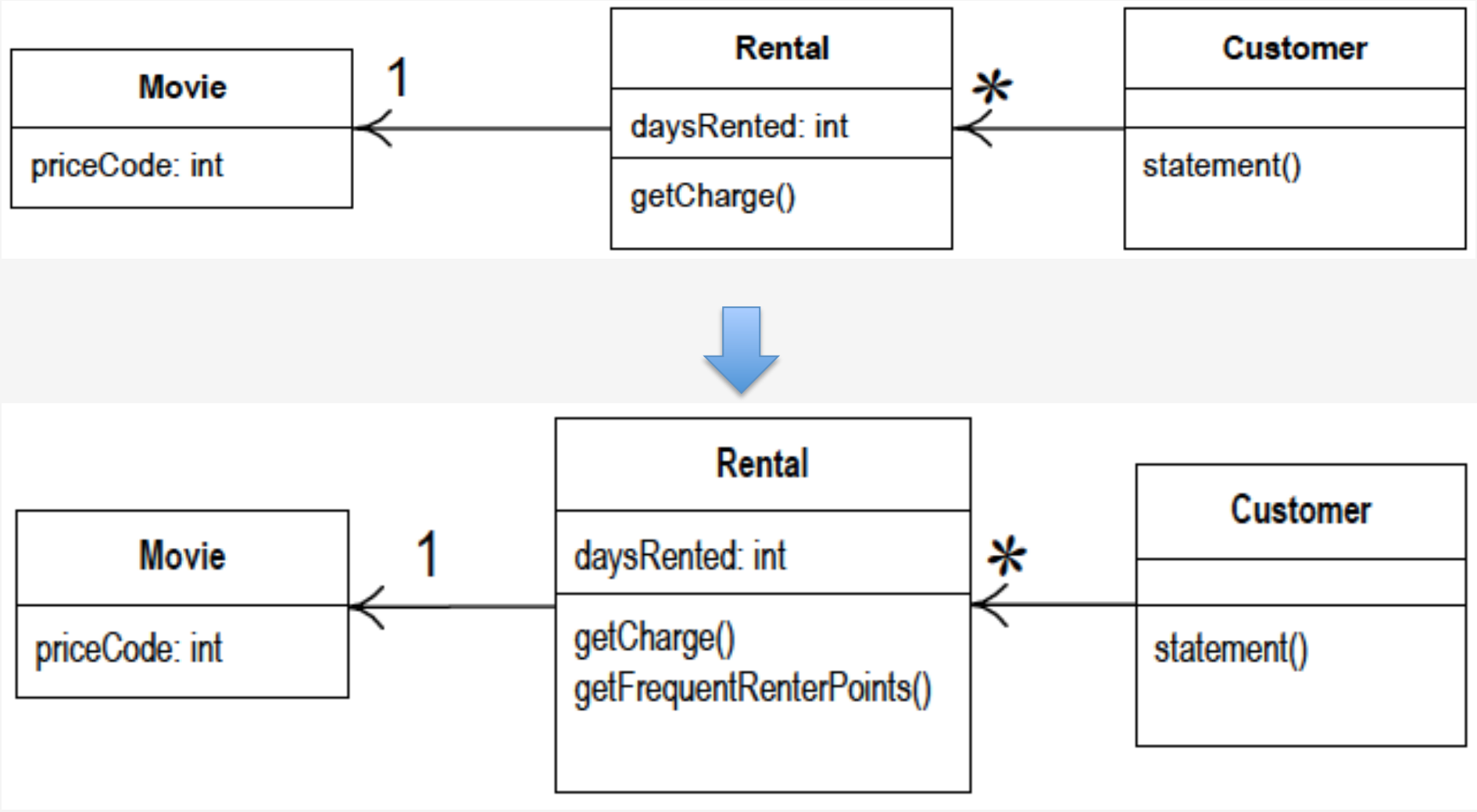
```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    // add frequent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

example after

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

example after



remove temps: example before

```
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()) {
      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;
  }
}
```

example after

```
class Customer { . . .
  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;
  }
}
```

example before

```
class Customer { . . .
  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;
  }
```

example after

```
class Customer { . . .
  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(getTotalChargeFrequentRenterPoints())+
      " 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;
  }
```

why refactoring?

- improves the design of software
- makes software easier to understand
- helps in finding bugs
- helps in programming faster

refactoring and design

- refactoring is a complement to design
- refactoring can be an alternative to upfront design
  1. the programmer codes, and then refactor it into shape (XP practice)
  2. software engineers do upfront design, but they don't try to find **the** solution; they need a reasonable solution
- refactoring can lead to simpler designs without sacrificing flexibility
- this makes the design process easier and less stressful

inline method

a method's body is just as clear as its name

put the method's body in to the body of its callers and remove the method

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return (_numberOfLateDeliveries>5)
}
```

```
int getRating() {
  return (_numberOfLateDeliveries>5) ? 2 : 1;
}
```

introduce explaining variable

there is a complicated expression

put the result of the expression (or its parts) in a temporary variable with an explaining name

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialised() && resize > 0 )
{
  // do something
}
```

```
final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized  = resize > 0;
if (isMacOs && isIEBrowser &&  wasInitialised() && wasResized )
{
  // do something
}
```

substitute algorithm

an algorithm can be replaced by a clearer one

replace the method's body with the new algorithm

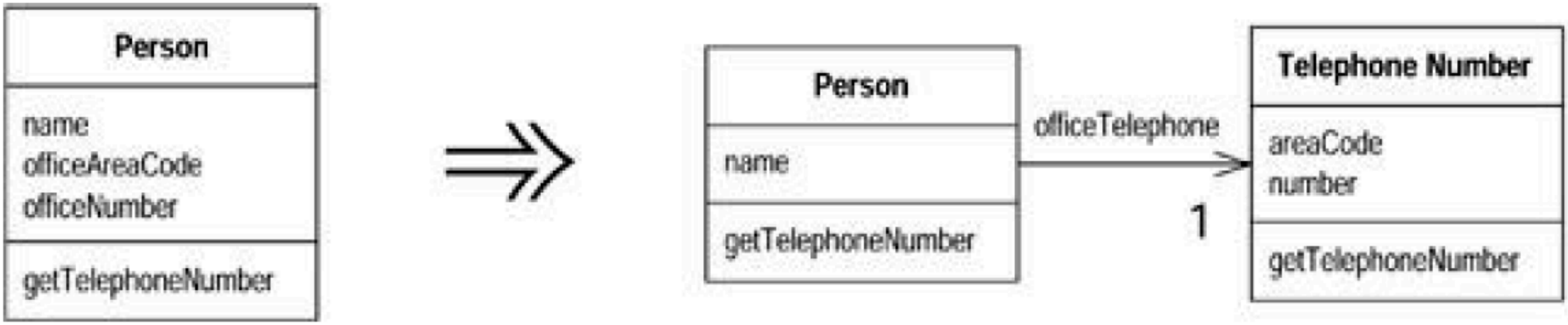
```
String foundPerson(String[] people) {
  for (int i=0; i < people.length; i++) {
    if (people[i].equals("Peter"))
      return "Peter"
    if (people[i].equals("John"))
      return "John"
    if (people[i].equals("Mike"))
      return "Mike"
  }
  return "";
}
```

```
String foundPerson(String[] people) {
  List candidates = Arrays.asList(new String[] {"Peter", "John", "Mike"});
  for (int i=0; i < people.length; i++)
    if (candidates.contains(people[i]))
      return people[i];
  return "";
}
```

extract class

one class does work that should be done by two

create a new class and move the relevant fields/ methods to the new class



- inline class is the reverse of extract class

bad code smells

- **Bloaters** are units of code that have increased to such proportions that they are hard to work with
- Usually these smells accumulate over time as the program evolves
- **Object-orientation abusers** smells are incomplete or incorrect application of OO programming principles
- **Change preventers smells** mean that if you need to change something in one place in your code, you have to make many changes in other places too.
- Program development becomes much more complicated and expensive as a result.

bad code smells

- A **dispensable** is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand
- A good example is “dead code”.
- **Couplers smells** contribute to excessive coupling between classes

exercise

refactoring the code of your project

1. identify smells in the code
2. apply refactorings
3. test the code to make sure the behavior is unchanged

key messages

- refactoring is a series of small steps, each of which changes the program's internal structure without changing its external behavior
- before you start refactoring, check that you have a solid suite of self-checking tests
- refactoring changes the programs in small steps

further reading

- Fowler; “Refactoring: Improving the design of existing code”; Addison-Wesley, 1999
- Refactoring [https://sourcemaking.com/refactoring]