CS420/520 — Object-oriented Programming

Refactoring

"Design is too important to be done only when we know nothing about the project"

Refactoring

- *Refactoring* is improving the design of existing code

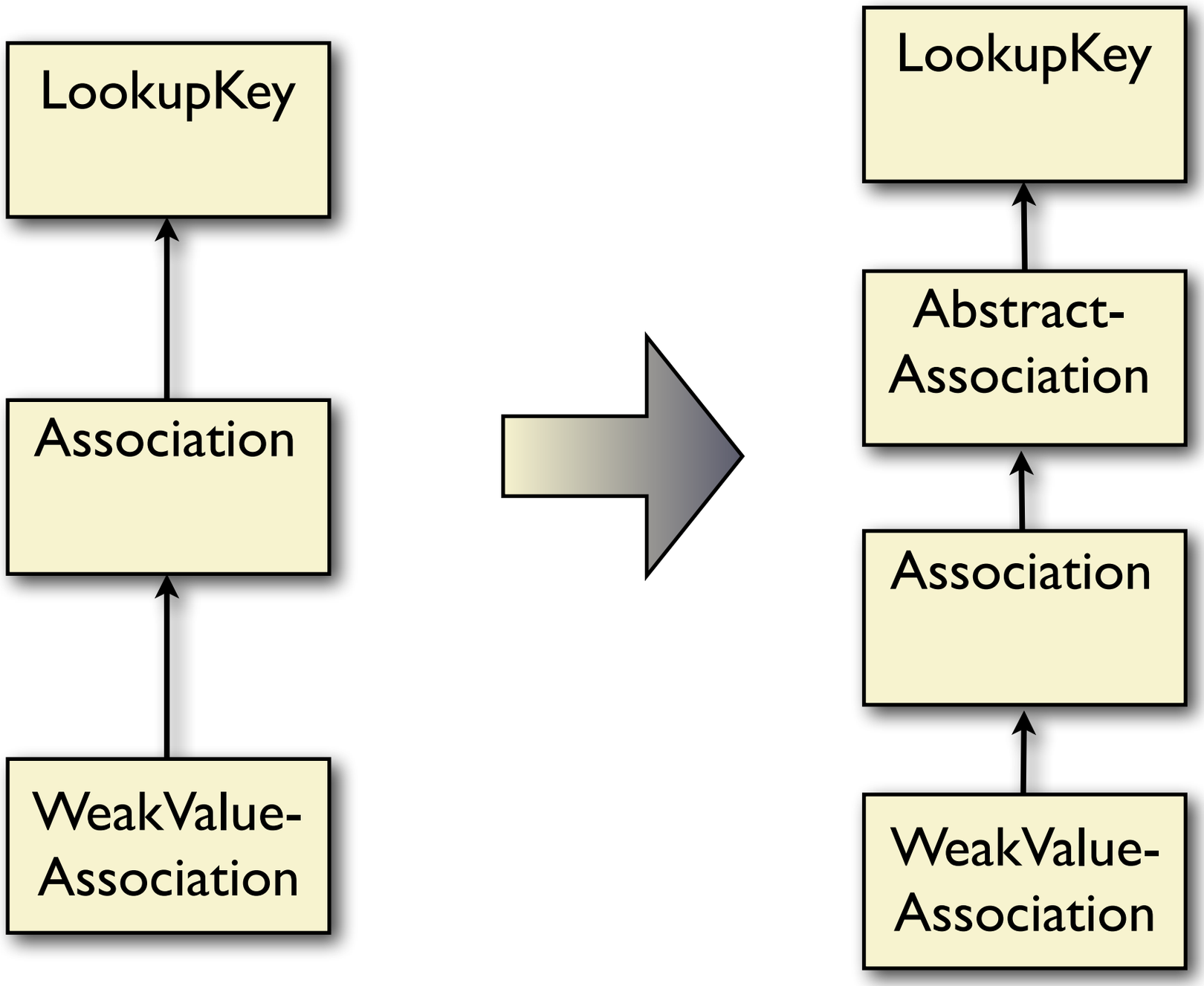
- Two choices:
  - Design up front, getting everything exactly right the first time, or
  - Design as you go, and be prepared to refactor

These slides based on materials by Don Roberts and John Brant

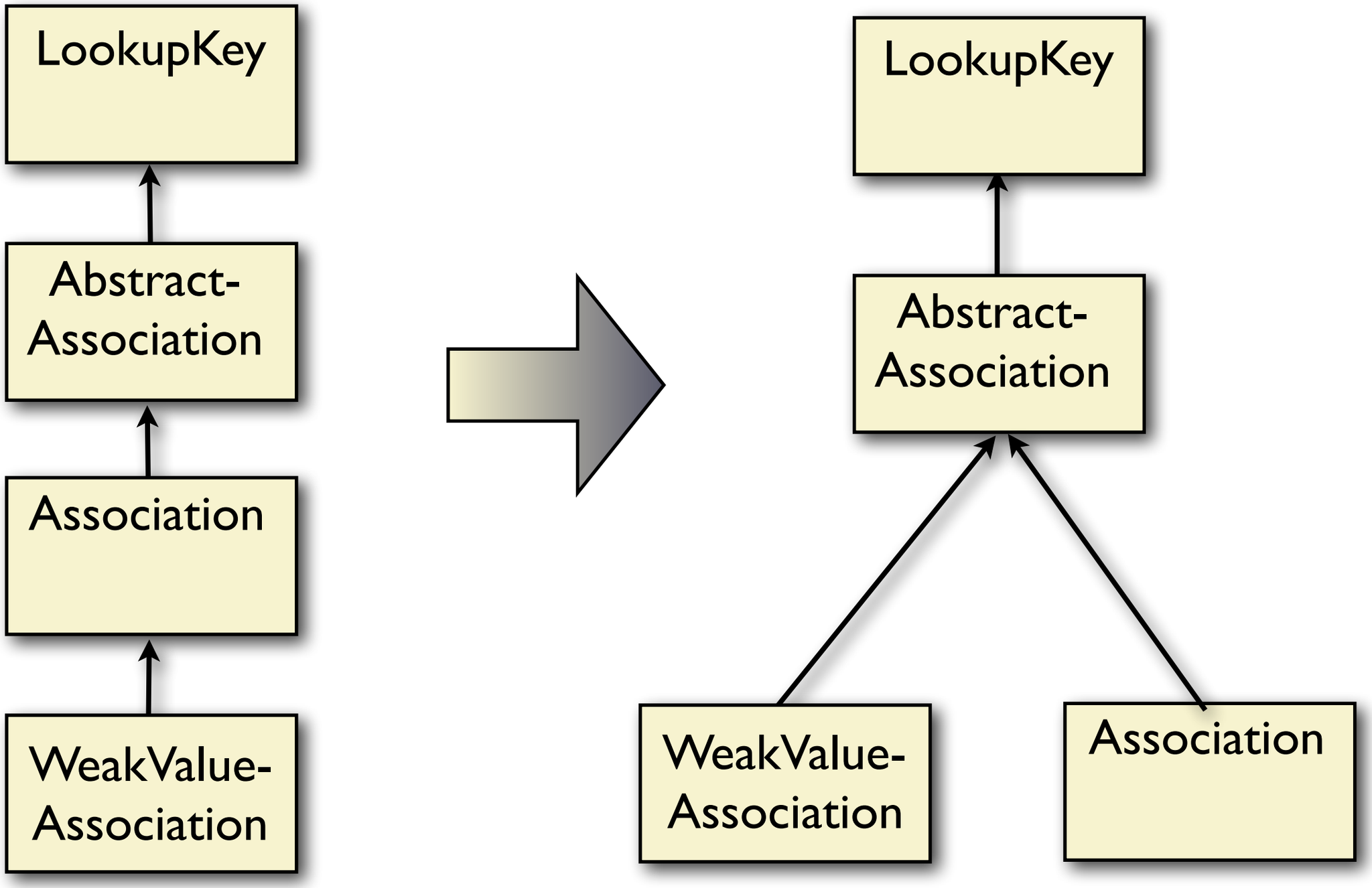
Software Maintenance

- Practically all “maintenance” is just continuing development
- Initial development is just “maintaining” a blank sheet of paper
- Software is never finished…
  - until it’s pried from the cold dead hands of its last user.

A Simple Refactoring: Add Empty Class



Another Refactoring …



So, what's the problem?

- Complexity
  - It's hard to understand what's there
- Fear
  - Changing what you don't understand is scary
- Errors
  - If you get it wrong, you break a working program

If it ain't broke, don't fix it.

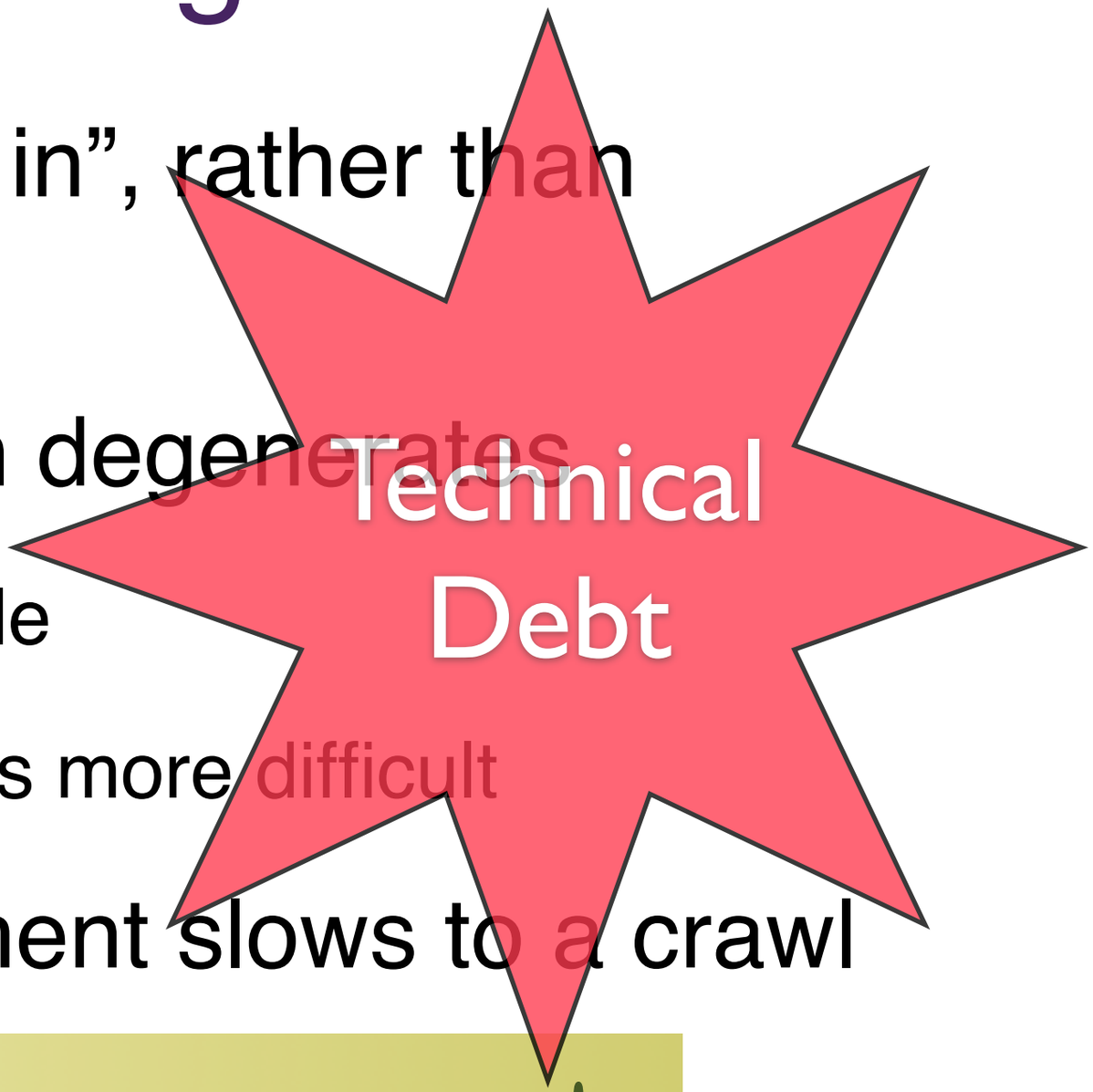
Schedule Pressure

- Every project is in a time crunch
- Refactoring can be time consuming
  - wouldn't it be better to put it off until after the next release?
- You are being paid to add *new* functionality

If it ain't broke, don't fix it.

Consequences of deferring refactoring

- Changes are “hacked in”, rather than designed
- Overall system design degenerates
  - Code becomes more brittle
  - The next change becomes more difficult
- The pace of development slows to a crawl



Don’t let this happen to your system!

The Refactoring Process

- Think about manipulating a mathematical expression:

$ax^2 + bx + c \Rightarrow axx + bx + c \Rightarrow (ax + b)x + c$

- Each step is semantics-preserving, so many small steps can be combined to have a large effect

Refactoring Example

- Chapter 1 of Fowler's book is an extended example.
- The initial code, written in Java, is an accounting system for a video rental store
  - Not a realistic example — too small

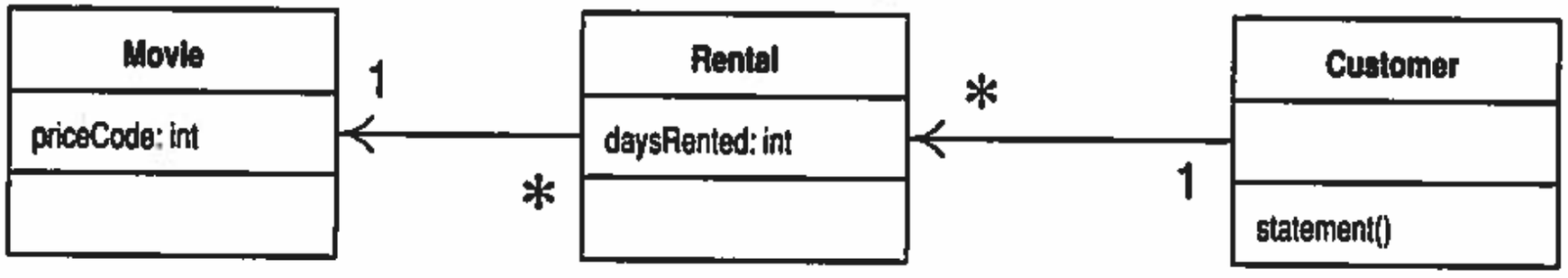


Figure 1.1 *Class diagram of the starting-point classes. Only the most important features are shown. The notation is Unified Modeling Language UML [Fowler, UML].*

Movie

Movie is just a simple data class.

```
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;
  };
}
```

```
class movie (title': String, priceCode': Number) {
    var priceCode is public := priceCode'
    method title { title' }
}
```

Rental

The rental class represents a customer renting a movie.

```
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 rental (movie':Movie, daysRented':Number) {
    method movie { movie' }
    method daysRented { daysRented' }
}
```

Customer

The customer class represents the customer of the store. Like the other classes it has data and accessors:

```
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;
  };
```

```
class customer (name':String) {
    def rentals = list.empty
    method addRental (arg:Rental) {
        rentals.add(arg)
    }
    method name { name' }
```

Customer also has the method that produces a statement. Figure 1.2 shows the interactions for this method. The body for this method is on the facing page.

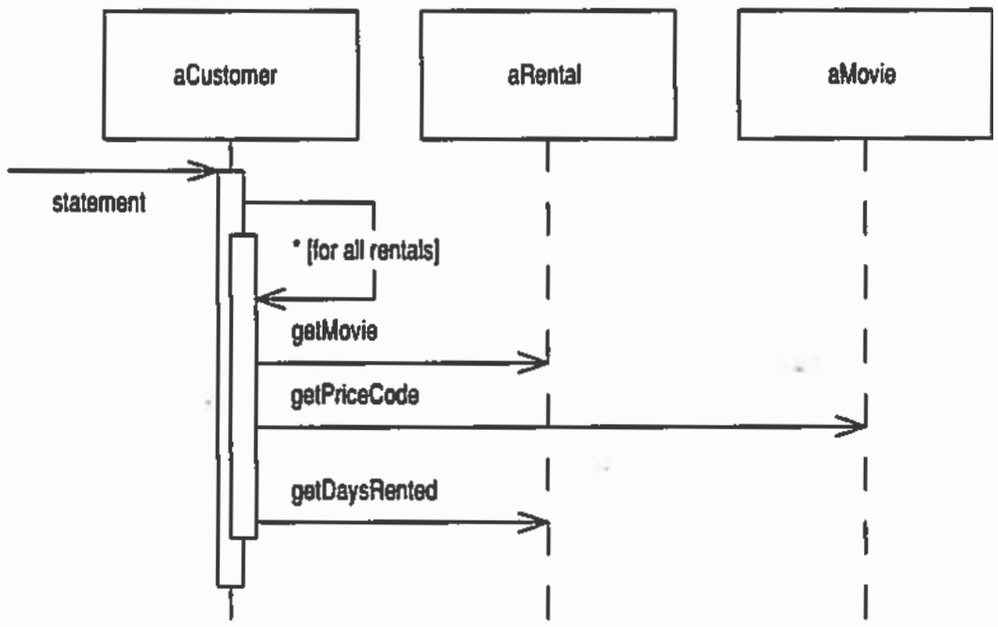


Figure 1.2 *Interactions for the statement method*

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

            //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;

            }

            // 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;

}
```

```
method statement {
    var totalAmount:Number := 0
    var frequentRenterPoints := 0
    var result := "Rental Record for {name}\n"
    for (rentals) do { each →
        var thisAmount := 0

        //determine amounts for each line
        match (each.movie.priceCode)
          case { kind.REGULAR →
            thisAmount := thisAmount + 2
            if (each.daysRented > 2) then {
                thisAmount := thisAmount + (each.daysRented - 2) * 1.5
            }
        } case { kind.NEW_RELEASE →
            thisAmount := thisAmount + each.daysRented * 3
        } case { kind.CHILDRENS →
            thisAmount := thisAmount + 1.5
            if (each.daysRented > 3) then {
                thisAmount := thisAmount + (each.daysRented - 3) * 1.5
            }
        }

        // add frequent renter points
        frequentRenterPoints := frequentRenterPoints + 1

        // add bonus for a two day new release rental
        if ((each.movie.priceCode == kind.NEW_RELEASE) &&
                (each.daysRented > 1)) then {
            frequentRenterPoints := frequentRenterPoints + 1
        }

        //show figures for this rental
        result := result ++ "\t{each.movie.title}\t{thisAmount}\n"
        totalAmount := totalAmount + thisAmount
    }
    //add footer lines
    result := result ++ "Amount owed is {totalAmount}\n"
    result := result ++ "You earned {frequentRenterPoints} frequent renter points"
    return result
}
```

Why Refactor?

We have a change that the users would like to make.

- First they want a statement printed in HTML so that the statement can be Web enabled and fully buzzword compliant.
- The users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make. They have a number of changes in mind. These changes will affect both the way renters are charged for movies and the way that frequent renter points are calculated
- As an experienced developer you are sure that whatever scheme users come up with, the only guarantee you're going to have is that they will change it again within six months.

The First Step in Refactoring

"Whenever I do refactoring, the first step is always the same. I … build a solid set of tests for that section of code. The tests are essential … even though I follow [a] refactoring [process that is] structured to avoid … introducing bugs. I'm still human and still make mistakes. Thus, I need solid tests."

Martin Fowler, *Refactoring*

Individual Refactorings

- Add Something:
  - Add field
  - Add temporary variable
  - Add Class variable
  - Add Class
  - Add methods

- Remove Something:
  - Remove field
  - Remove temporary variable
  - Remove Class variable
  - Remove Class
  - Remove methods

- Rename Something:
  - Rename field
  - Rename temporary variable
  - Rename Class variable
  - Rename Class
  - Rename methods (see next slide)

- Move Something:
  - Move field up or down
  - Move temp to inner/outer scope
  - Move class variable up or down
  - Move method to component
  - Move field to component
  - Change superclass

Method-level Refactorings

- Method Renamings
  - Simple rename
  - Permute arguments
  - Add argument
  - Remove argument

- Introductions
  - Extract code into method
  - Extract code into temporary variable

- Eliminations
  - Inline method
  - Inline temporary

Safe Refactoring

- Use tests
  - tests should pass before and after refactoring
- Use a refactoring tool if it's available
  - Smalltalk Refactoring Browser
  - Plugins for Java in Eclipse
- Take small steps, testing between each step

Code Smells

- Develop a “nose” for code
  - Does the code smell bad?
- What bad smells have you seen in others’ code?

Some smells that I have known

Code violates the "once and only once" rule

- code does not say it *at all*
- code says it *twice*, *thrice*, … *fifteen times*!

Methods are too large

- Why is this a problem?
  - methods are the smallest unit of overriding
  - statements in a method should be at same level of abstraction

Methods in the wrong class

- if a method does not refer to **self**, it is probably in the wrong class
  - implicit **self** counts as referring to self
- check the parameters
- However:
  - there are “utility methods” that have no natural home

“Feature Envy”

- method over-uses accessors (getters and setters) of another object
- can the method be moved into the other object?
  - sometimes only *part* of the method should be moved
  - extract method into component

The “God” class

- a large class with many methods and many fields
- can you partition the methods and the fields that they access?
- turn each partition into new class
  - large class becomes composition of smaller classes

Field not always used

- Some instances use a particular field, others don't
- Create two or more subclasses with the *right* fields
- Is a field used only during a certain operation?
  - "operation" spans more than one method
  - consider using a method object

Co-occurring Parameters
(a.k.a Data Clumps)

- if the same pair (or triplet) of parameters is passed to several methods:
- perhaps they represent an abstraction that should be captured in an object?
  - e.g., *x* and *y* should be grouped into a point object
  - e.g., *list* and *index* should be grouped into an iterator object
- once the object exists, you will often find it natural to add *behavior*

Comments

- Most comments are written to compensate for poorly written code!
  - if you feel that your code needs *explaining*, consider *refactoring* it instead

Original Code

```
initialize
        | w button |

        super initialize.
        self layoutPolicy: TableLayout new.
        self listDirection: #leftToRight.
        self layoutInset: 2.
        self borderWidth: 0.
        self hResizing: #shrinkWrap.
        self vResizing: #shrinkWrap.
        self color: Color gray.

        w := TheWorldMenu new

world: World project:
                (World project ifNil: [Project current])
        hand: World primaryHand.

    button := LaunchButtonMorph new.
        button label: 'Browser';
        actionSelector: #openBrowser;
target: Browser;
        actWhen: #buttonUp.
        self addMorph: button.

button := LaunchButtonMorph new.
        button label: 'Workspace';
        actionSelector: #openWorkspace;
target: w;
        actWhen: #buttonUp.
        self addMorph: button.

button := LaunchButtonMorph new.
        button label: 'Transcript';
        actionSelector: #openTranscript;
        target: w;
        actWhen: #buttonUp.
        self addMorph: button.

button := LaunchButtonMorph new.
        button label: 'Change Sorter';
        actionSelector: #openChangeSorter2;
target: w;
        actWhen: #buttonUp.
        self addMorph: button.

button := LaunchButtonMorph new.
        button label: 'File List';
        actionSelector: #openFileList;
        target: w;
actWhen: #buttonUp.
        self addMorph: button
```

Add comments and explaining names

```
initialize
        | w browserButton workspaceButton
transcriptButton changeButton fileListButton |

        super initialize.

    "initialize layout"
self layoutPolicy: TableLayout new.
        self listDirection: #leftToRight.
        self layoutInset: 2.
        self borderWidth: 0.
        self hResizing: #shrinkWrap.
        self vResizing: #shrinkWrap.
        self color: Color gray.

        w := TheWorldMenu new

world: World project:
                (World project ifNil: [Project current])
        hand: World primaryHand.

    "initialize buttons"
     browserButton := LaunchButtonMorph new.
        browserButton label: 'Browser';
        actionSelector: #openBrowser;
target: Browser;
        actWhen: #buttonUp.
        self addMorph: browserButton.

workspaceButton := LaunchButtonMorph new.
        workspaceButton label: 'Workspace';
        actionSelector: #openWorkspace;
target: w;
        actWhen: #buttonUp.
        self addMorph: workspaceButton.

transcriptButton := LaunchButtonMorph new.
        transcriptButton label: 'Transcript';
        actionSelector: #openTranscript;
        target: w;
        actWhen: #buttonUp.
        self addMorph: transcriptButton.

changeButton := LaunchButtonMorph new.
        changeButton label: 'Change Sorter';
        actionSelector: #openChangeSorter2;
target: w;
        actWhen: #buttonUp.
        self addMorph: changeButton.

fileListButton := LaunchButtonMorph new.
        fileListButton label: 'File List';
        actionSelector: #openFileList;
        target: w;
actWhen: #buttonUp.
        self addMorph: fileListButton
```

Composed Method

```
initialize
        super initialize.
        self initializeLayout.
        self initializeButtons
```

```
initializeLayout
        self layoutPolicy: TableLayout new.
        self listDirection: #leftToRight.
        self layoutInset: 2.
        self borderWidth: 0.
        self hResizing: #shrinkWrap.
        self vResizing: #shrinkWrap.
        self color: Color gray.
```

```
initializeButtons
        | w |
        w := TheWorldMenu new
                world: World
                project: (World project ifNil: [Project current])
                hand: World primaryHand.
        self addAButton: 'Browser' sending: #openBrowser to: Browser.
        self addAButton: 'Workspace' sending: #openWorkspace to: w.
        self addAButton: 'Transcript' sending: #openTranscript to: w.
        self addAButton: 'Change Sorter' sending: #openChangeSorter2 to: w.
        self addAButton: 'File List' sending: #openFileList to: w
```

```
addAButton: label sending: s to: target
        | button |
        button := LaunchButtonMorph new.
        button label: label;
                actionSelector: s;
                target: target;
                actWhen: #buttonUp.
        self addMorph: button
```

Nested Conditionals

- Message send = procedure call + case selection
  - use this to eliminate explicit conditionals
  - the goal: adding new cases does not require changing existing code
- *e.g.*, instead of testing isEmpty or isNil, consider a *separate object* to represent the Empty or Nil case
  - The Null Object Pattern (http://www.cs.oberlin.edu/~jwalker/refs/woolf.ps)

Nested Conditionals (cont.)

- Early returns are often better than nested conditionals.

```
method totalDue {
    // Answer the total owed
    if (self.cart.isEmpty) then { return 0 }
    var result := 0
    for (cart.items) do { each →
        result := result + (each.cost * each.count)
    }
    return result
}
```

- Is there a need for the test at all?

Nested Conditionals (cont.)

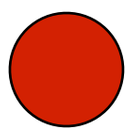
- If conditional involves a test of the object's class, move the method to that class
  - self class = … or
  - isKindOf:

```
AbstractSound >>loudness: aNumber
        "Initialize my volume envelopes and initial volume. …"
        | vol |
        vol := (aNumber asFloat max: 0.0) min: 1.0.
        envelopes do: [:e | (e isKindOf: VolumeEnvelope) ifTrue: [e scale: vol]].
        self initialVolume: vol.
```

Strategies for Refactoring

1. Extend then Refactor
2. Refactor then Extend
3. Debug then Refactor
4. Refactor then Debug
5. Refactor for Understandability

Extend then Refactor

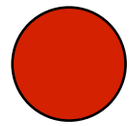
- test fails 
- hack in a change to make the test pass 
  - e.g., copy and paste a method, and then edit the new method.
- test passes, but you are not done yet! 
  - eliminate redundancy

Coding is like mountain climbing: getting the green light is like reaching the summit

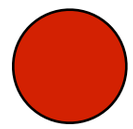
Refactor then Extend

- It seems awkward to implement a new feature
- Refactor design to make the change easy
- add a test for the feature
- add the feature



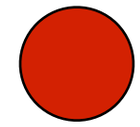


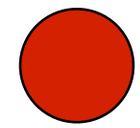

Debug then Refactor

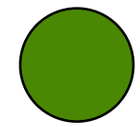
- Find the bug 
- Fix the bug 
- Refactor to make the bug obvious, *e.g.*, 
  - extract method and give it an *explaining name*
  - rename method or temp
  - extract expression to temporary variable
    - eliminate “magic numbers”

Refactor then Debug

- Suppose that you can't find the bug?
  - Refactoring preserves bad behavior too!
- Simplify complex method
- Fix the bug







Refactor for Understandability

- What was obvious when a method was written isn't always obvious a day later!
  - use composed method (Beck p. 21)
  - use intention revealing selectors (Beck p. 49)
  - use explaining temporary variable (Beck p. 108)
  - don't worry about performance
    - "clever" code is usually dumb

The Loan Metaphor

"Quick and dirty" coding is like taking out a loan

Living with the bad code is like paying interest

Refactoring your code is like paying off the loan

— — — —

- Some debt is OK, in fact necessary, to grow a business
- Too much debt is unhealthy: it will eventually kill you

"Technical Debt"

must be paid off

Listen to your Code

- If something seems difficult or awkward, refactor to make it easy
- Let the program tell you where it needs to be fixed
  - Does the code speak to you? Does it smell?
- If you copy and paste, you *probably want to* refactor to remove the duplication

Do you know all of the refactorings?

List of Refactorings

List of Refactorings