

***Software
Technology
Group***

TU Darmstadt | FB Informatik

Software Engineering Design

8. Refactoring

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

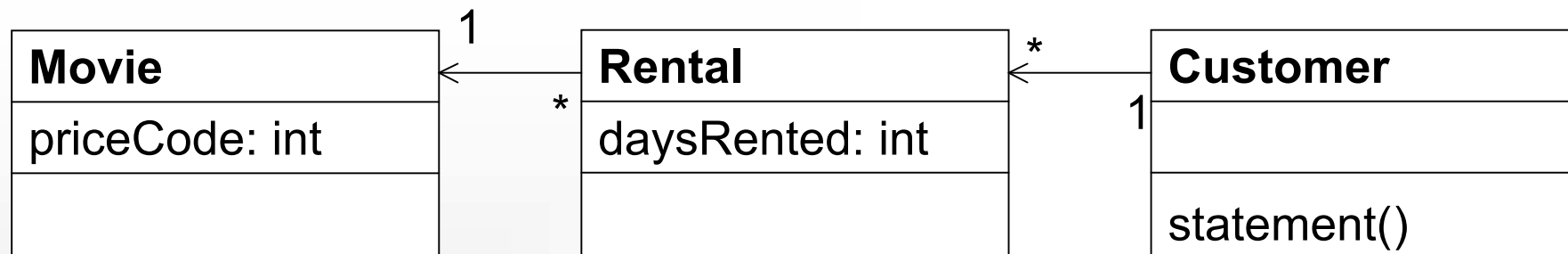
Dipl.-Ing. Michael Haupt

Agenda

- getting a feeling about what refactoring is
- what is refactoring and its role in the software development process
- some selected refactorings

An Example

- a program for calculating and printing a statement of a customer's charges at a video store
- there are three different kinds of movies: **CHILDRENS**, **NEW_RELEASE**, and **REGULAR**
- the type of a movie determines the way the charge for the movie is calculated



- beside the charge, also frequent renter points are calculated during the preparation of the statement

An Example

```
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; }
}
4
```

An Example

```
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; }  
}
```

An Example

```
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; }

    // more: see next slides
}
```

An Example

```
// class Customer continued
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();

    // add header notes
    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;
```

An Example

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

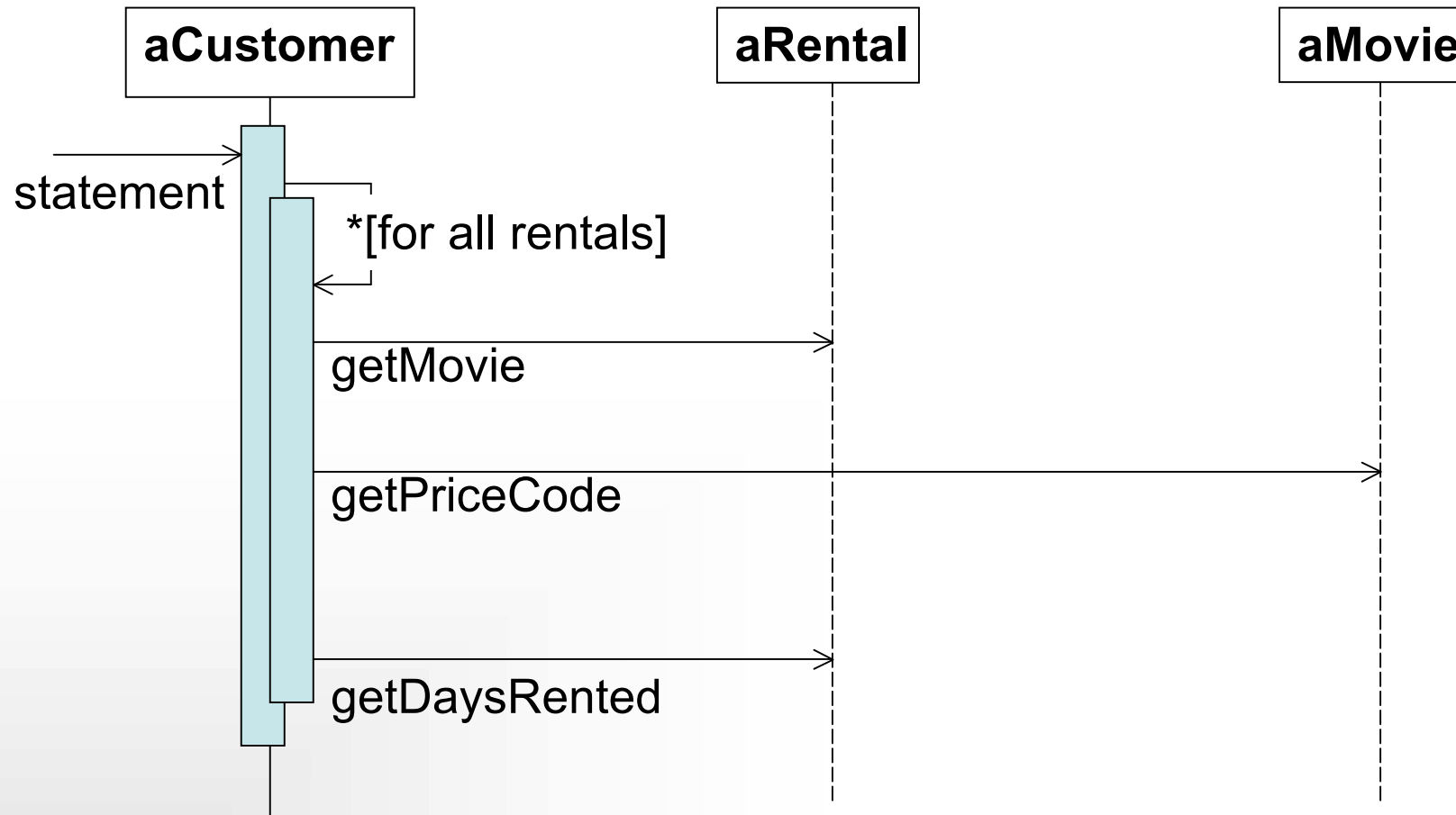
An Example

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

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

return result;
} // end statement
```

An Example



Brief Evaluation

- What are your impressions about the code of the example?
- Is it well designed?
- Is it "object-oriented"?

1st Improvement...?

```
// class Customer continued
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();

    // add header notes
    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:
                ...
                break;
            ...
        }
    }
}
```

1st Improvement

Extract Item Charge Calculation

```
//class Customer...
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;
}
```

1st Improvement

Extract Item Charge Calculation

```
// 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
        frequentRenterPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
            && each.getDaysRented() > 1)
            frequentRenterPoint++;
        ...
    }
}
```

2nd Improvement...?

```
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;
}
```

2nd Improvement

Better Names

```
public double amountFor(Rental aRental) { // was: each
    double result = 0; // was: thisAmount
    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;
}
```

Format of Refactorings

- **Name:**
 - important to build a vocabulary of refactorings
- **Summary:**
 - shortly describes the situation in which you need the refactoring and what the refactoring does
 - helps finding a refactoring more quickly
- **Motivation:**
 - why the refactoring should be done
 - situations in which the refactoring shouldn't be done
- **Mechanics:**
 - concise, step-by-step description of how to carry out the refactoring
- **Examples:**
 - illustrate how the refactoring works

Extract Method

- 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);  
}
```

Extract Method: Motivation

- a method is too long, or you need comments to understand it?
 - a good sign that extract method is a good thing to do
- short well-named methods should be preferred
 - increased chances that the code will be used by other methods
 - allows higher-level methods to be read more like a series of comments
 - overriding is easier with fine-grained methods
- good naming is a key issue

Extract Method: Mechanics

- create a new method and name it after the intention of the method
 - "methods as goals"
- copy extracted code from the source into the new method
- scan extracted code for references to any variables that are local in the scope to the source method
 - these are local variables and parameters to the method
- if any temps are used only within extracted code, declare them as temporary in the target method

Extract Method: Mechanics

- if one of the local-scope variables is modified by extracted code, see whether you can use the extracted method as a query and assign the result to the variable concerned
- pass into the target method as parameters local-scope variables read from extracted code
- compile when you have dealt with all local-scope variables
- replace extracted code in the target method with a call to the new method
- if you have moved any temps over to target that are declared outside the extracted code, remove them
- **compile and test**

Is the Method in the Right Place?

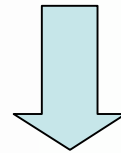
```
// class Customer...
public double amountFor(Rental aRental) {
    double result = 0;
    switch(aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if(aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented()-2) * 1.5;
            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;
}
```

3rd Improvement

Moving amountFor () to Rental

- remove the parameter... and give it a better name

```
// class Customer ...  
public double amountFor(Rental aRental) {  
    double result = 0;  
    switch (aRental.getMovie().getPriceCode()) {  
        ...  
    }  
}
```



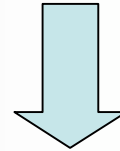
```
// class Rental ...  
public double getCharge() {  
    double result = 0;  
    switch (aRental.getMovie().getPriceCode()) {  
        ...  
    }  
}
```

3rd Improvement

Moving amountFor () to Rental

- change method calls

```
// class Customer ...  
public double getCharge() {  
    double result = 0;  
    switch(aRental.getMovie().getPriceCode()) {  
        ...  
    }  
}
```



```
// class Rental ...  
public double getCharge() {  
    double result = 0;  
    switch(getMovie().getPriceCode()) {  
        ...  
    }  
}
```

3rd Improvement

Moving amountFor () to Rental

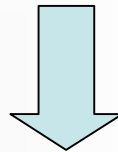
```
// class Rental ...
public double getCharge() {
    double result = 0;
    switch(getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if(getDaysRented() > 2)
                result += (getDaysRented()-2) * 1.5;
            break;
        case Movie.NEW_RELEASE: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

3rd Improvement

Moving amountFor () to Rental

- **test it!**

```
// class Customer...  
public double amountFor(Rental aRental) {  
    double result = 0;  
    switch(aRental.getMovie().getPriceCode()) {  
        ...  
    }  
}
```



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

3rd Improvement

Moving amountFor () to Rental

- ...and finally change statement ()

```
public String statement() {  
    ...  
    while (rentals.hasMoreElements()) {  
        double thisAmount = 0;  
        Rental each = (Rental) rentals.nextElement();  
  
        thisAmount = each.getCharge();  
  
        ...  
    }  
}
```

4th Improvement...?

Get rid of temps
like `thisAmount`

```
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();
        ...
        // show figures for this rental
        result += ... + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }
    ...
}
```

Temps cause a lot of parameters to be passed around.
You can easily lose their track.

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression.
- Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

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



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

Replace Temp with Query

- temps are useful only within their method, and thus they encourage long, complex routines
- in our example we have two temps that are used to calculate two totals from rentals
- these totals might be needed in several contexts, e.g., for printing the statement in both ASCII and HTML
- substitute temporary variables with query methods
 - **queries are accessible from any method in the class**
 - **encourage a cleaner design without long, complex statements**

4th Improvement

Get Rid of `thisAmount`

```
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();
        ...
        // show figures for this rental
        result += ... + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    ...
}
```

5th Improvement...?

```
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)  
            frequentRenterPoint++;  
        // show figures for this rental  
        result += ... + String.valueOf(each.getCharge()) + "\n";  
        totalAmount += each.getCharge();  
    }  
    ...  
}
```

move to Rental

5th Improvement

Move frequentRenterPoints

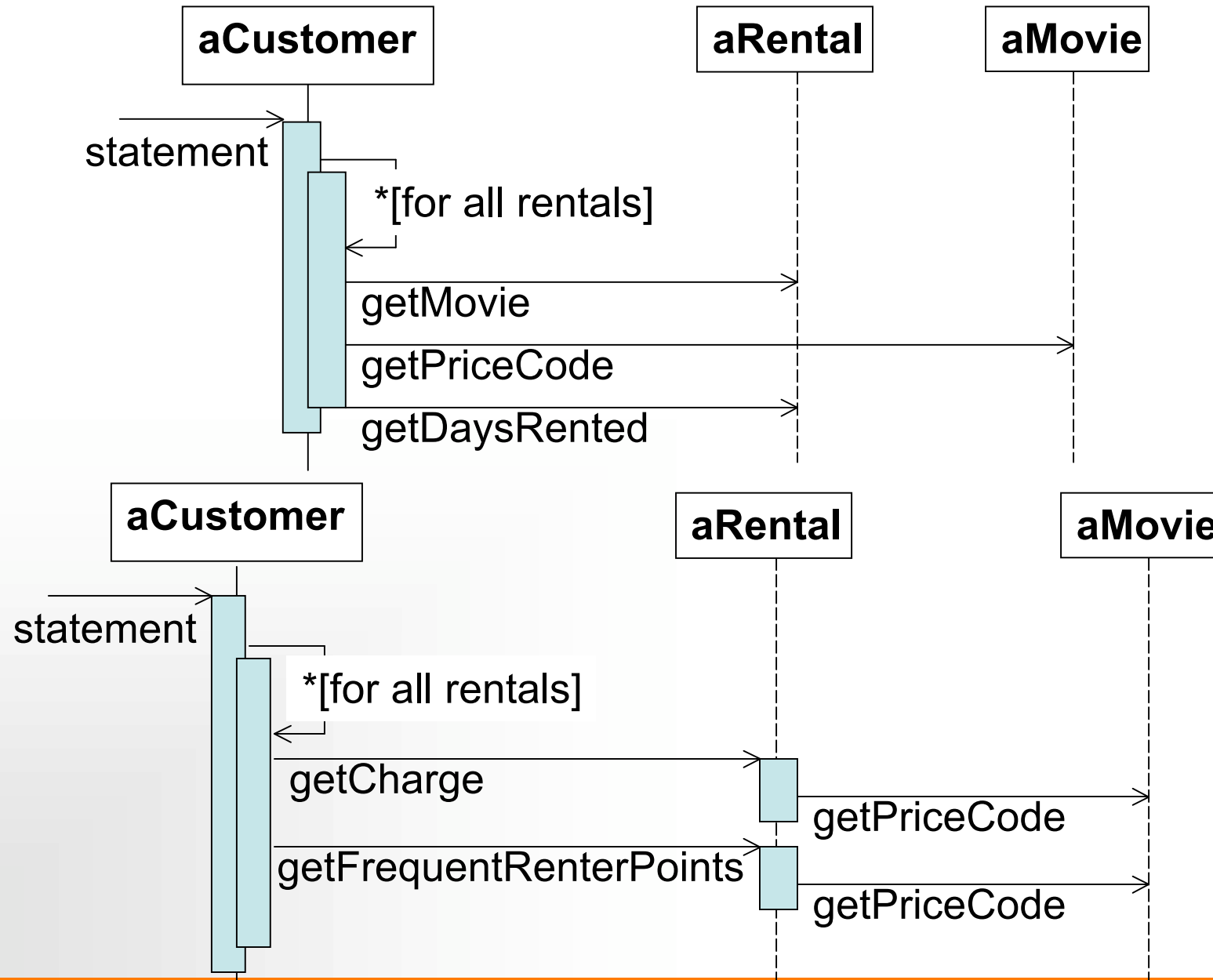
```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
    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

5th Improvement

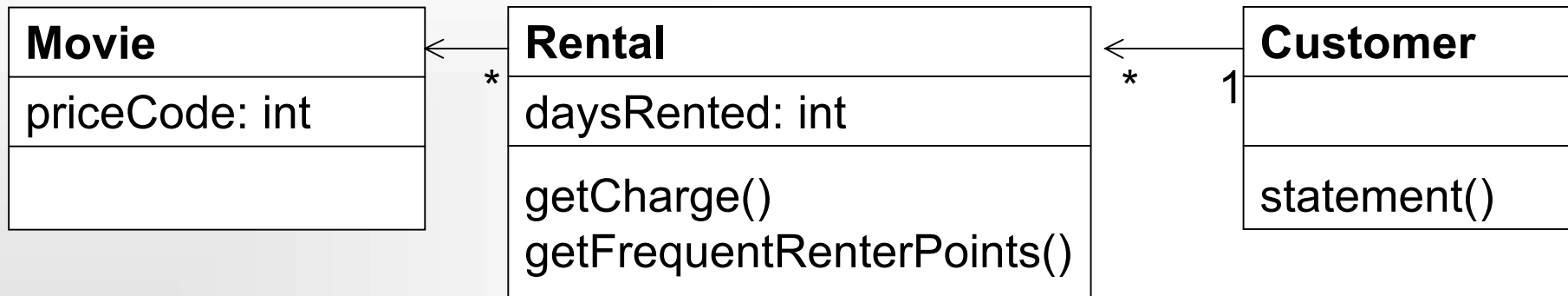
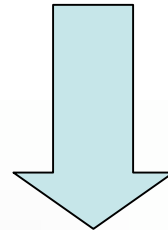
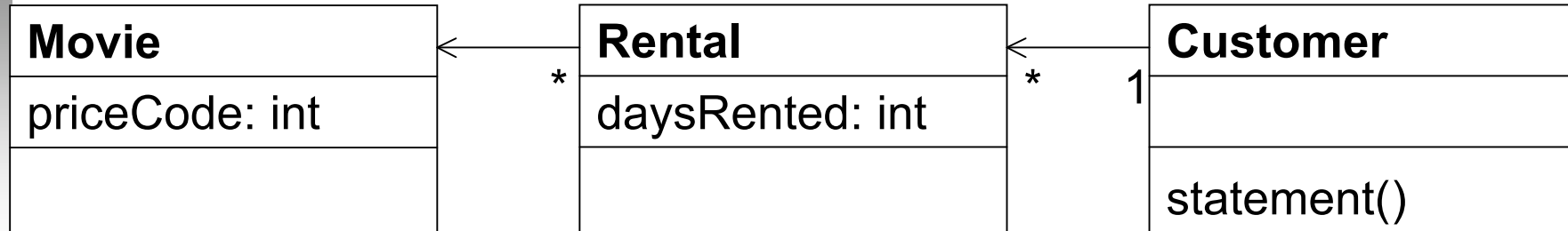
Move frequentRenterPoints

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

The Example So Far



The Example So Far



6th Improvement...?

What about getting rid
of the remaining temps?

```
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 += ... + String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }
    // add footer lines
    result += "Amount is " + String.valueOf(totalAmount) + "\n";
    result += "You earned "
        + String.valueOf(frequentRenterPoints)
        + " frequent renter points";
    return result;
} // end statement
```

6th Improvement

Getting Rid of totalAmount

```
public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
        // show figures for this rental
        result += ... + String.valueOf(each.getCharge()) + "\n";
    }
    // add footer lines
    result += "Amount is "
        + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned "
        + String.valueOf(frequentRenterPoints)
        + " frequent renter points";
} // end statement
```

6th Improvement

Getting Rid of `totalAmount`

```
public double getTotalCharge() {  
    double result = 0;  
    Enumeration rentals = _rentals.elements();  
    while (rentals.hasMoreElements()) {  
        Rental each = (Rental) rentals.nextElement();  
        result += each.getCharge();  
    }  
} // end getTotalCharge
```

- `totalAmount` is being assigned within the loop
- so the loop has to be repeated

6th Improvement

Getting Rid of frequentRenterPoints

```
// class Customer

public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // show figures for this rental
        result += ... + String.valueOf(each.getCharge()) + "\n";
    }
    // add footer lines
    result += "Amount is "
        + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned "
        + String.valueOf(getTotalFrequentRenterPoints())
        + " frequent renter points";
} // end statement

40
```

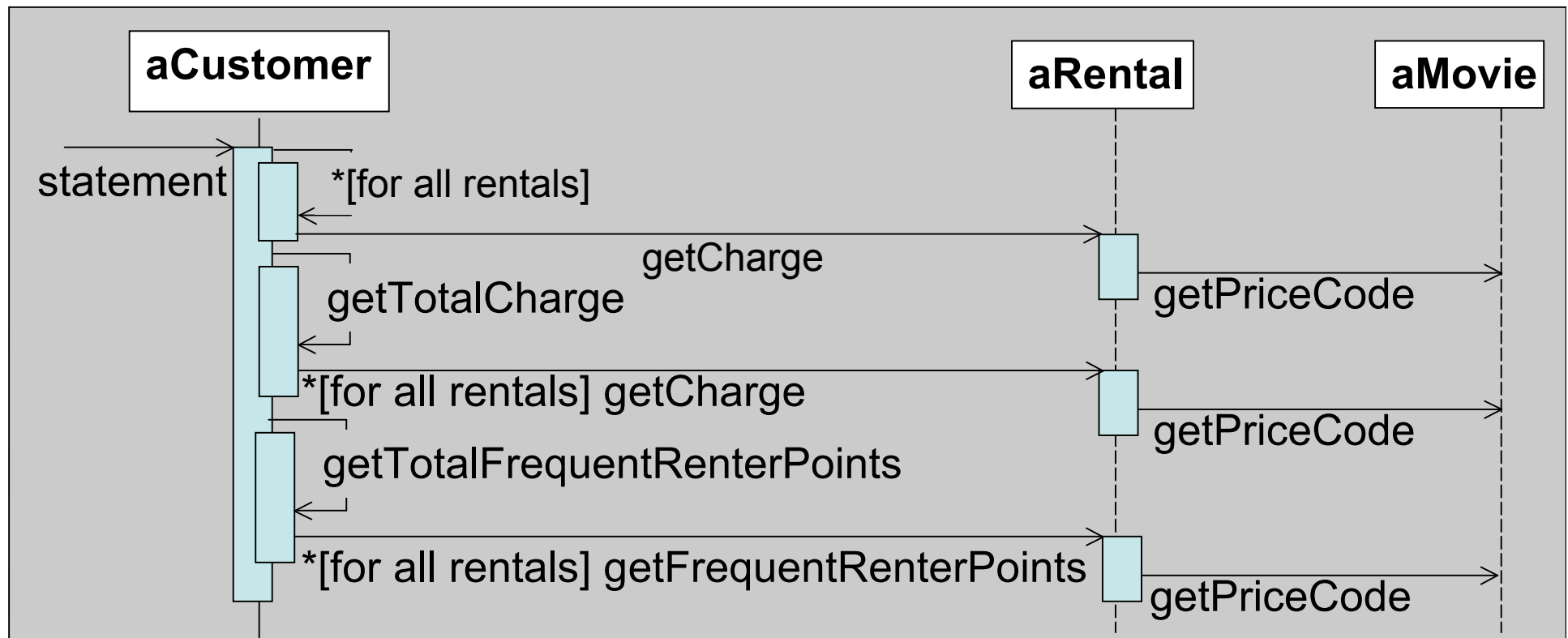
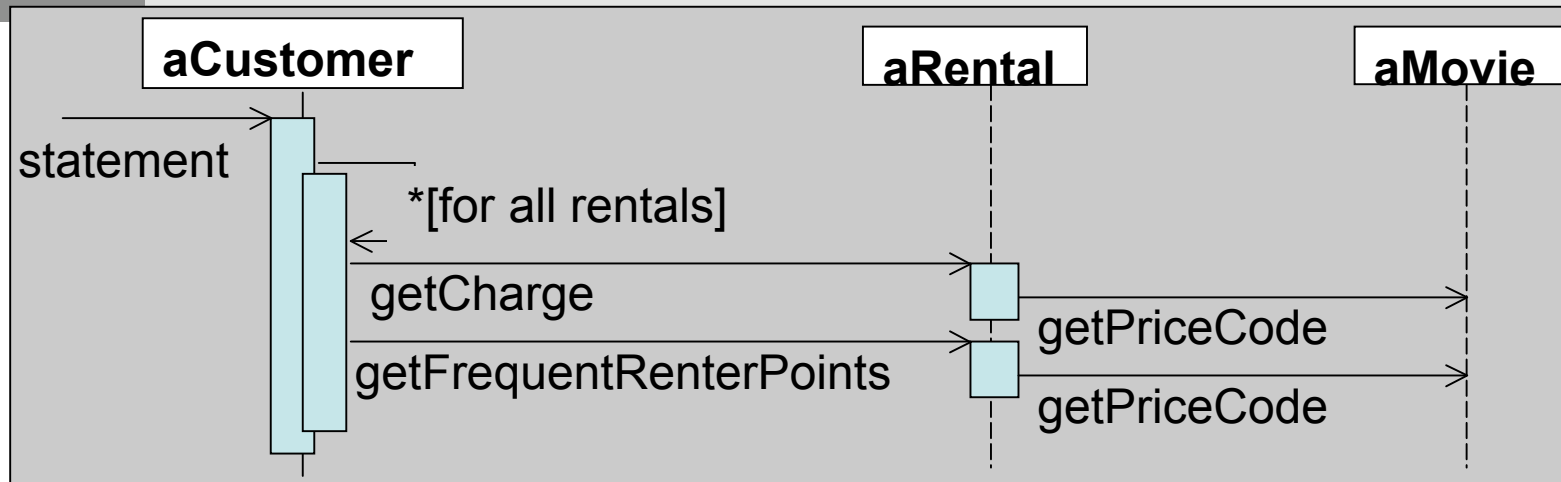
6th Improvement

Getting Rid of `frequentRenterPoints`

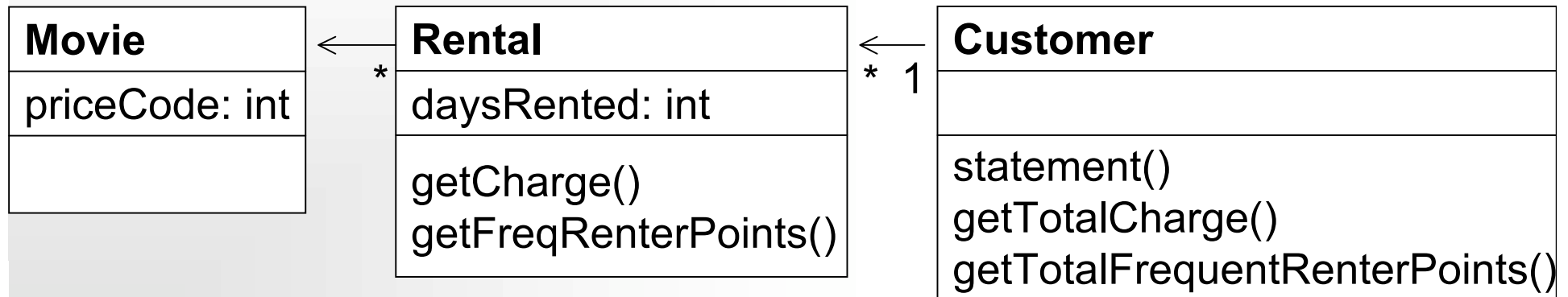
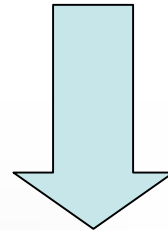
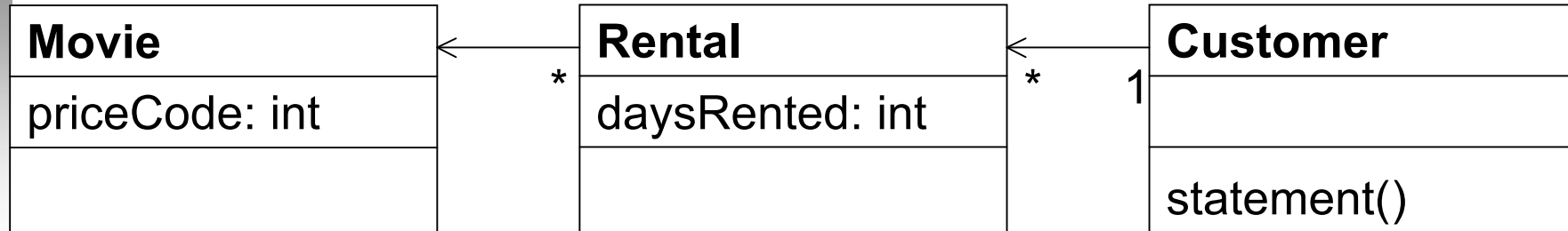
```
public int getTotalFrequentRenterPoints() {  
    double result = 0;  
    Enumeration rentals = _rentals.elements();  
    while (rentals.hasMoreElements()) {  
        Rental each = (Rental) rentals.nextElement();  
        result += each.getFrequentRenterPoints();  
    }  
} // end getTotalCharge
```

- `frequentRenterPoints` is being assigned within the loop
- so the loop has to be repeated

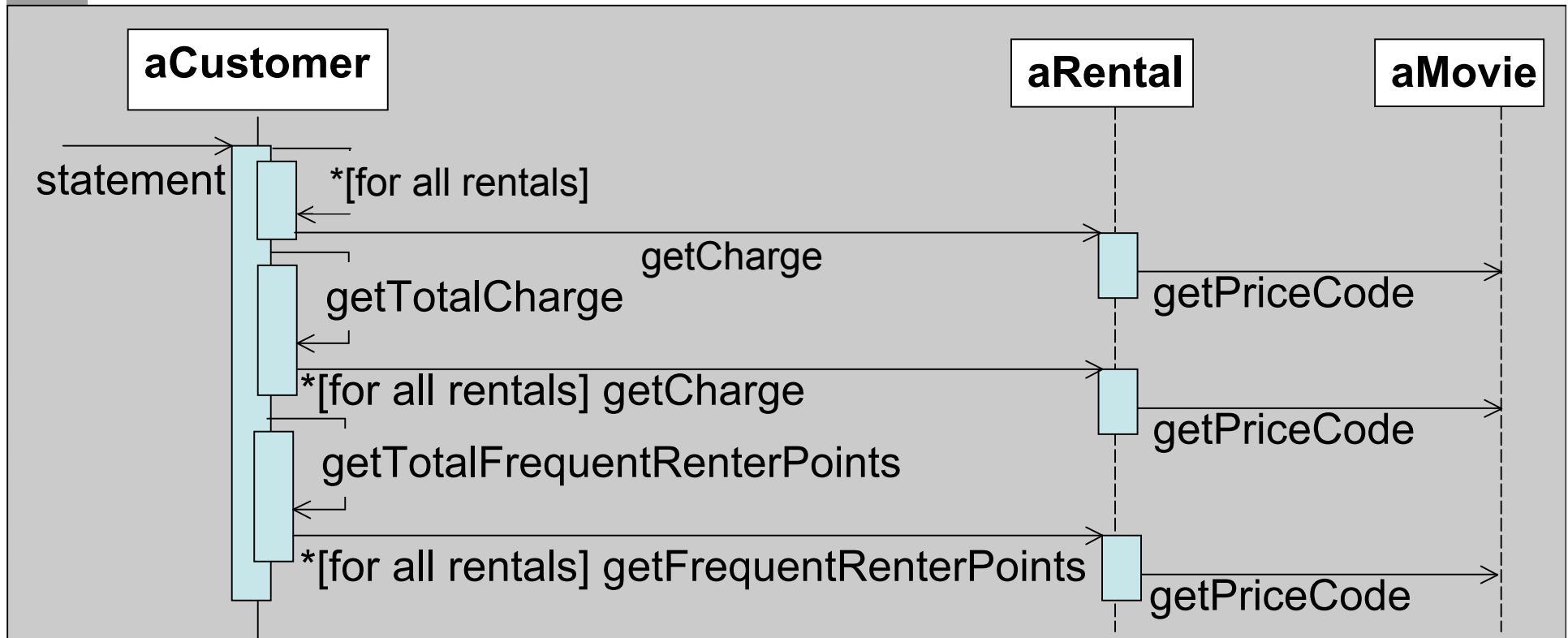
The Example so Far



The Example so Far



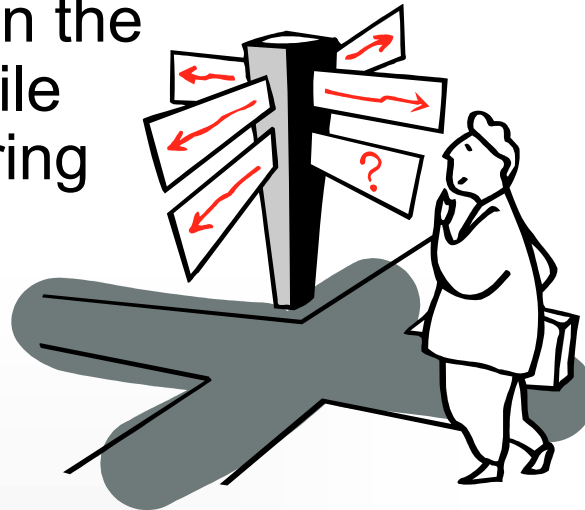
Wait a Minute...



What about **performance?!**
Before we executed the while loop once
The new code executes it three times!
A while loop **might take a long time!**

Refactoring *and* Performance?

Refactoring slows down software in the short term – while you are refactoring



It helps writing faster software in the long term because it makes software easy to tune during optimisations

Changing the Hat

From Refactoring to Adding Functions...

```
// class Customer
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + getName()
                  + "<\EM><\H1>\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // show figures for this rental
        result += ... + String.valueOf(each.getCharge())
                  + "<BR>\n";
    }
    // add footer lines
    result += "<P> you owe <EM>"
             + String.valueOf(getTotalCharge())
             + "<\EM><P>\n";
    result += "You earned <EM>"
             + String.valueOf(getTotalFrequentRenterPoints())
             + " frequent renter points<P>";
} // end statement
```

Changing the Hat

From Refactoring to Adding Functions...

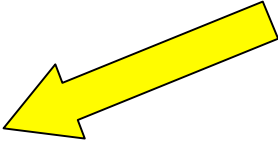
- I could reuse the calculating methods for the html statement
- however, I still did some copy-and-paste from the ASCII version
- I can further improve by factoring out methods for header and footer

Any idea how?

7th Improvement...?

```
// 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.CHIDRENS:
            result += 1.5;
            if(getDaysRented() > 3)
                result += (getDaysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

If switch, then at least on the own data



7th Improvement

Moving Charge Calculation

```
// class Movie...
public 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;
}
```

7th Improvement

Moving Charge Calculation

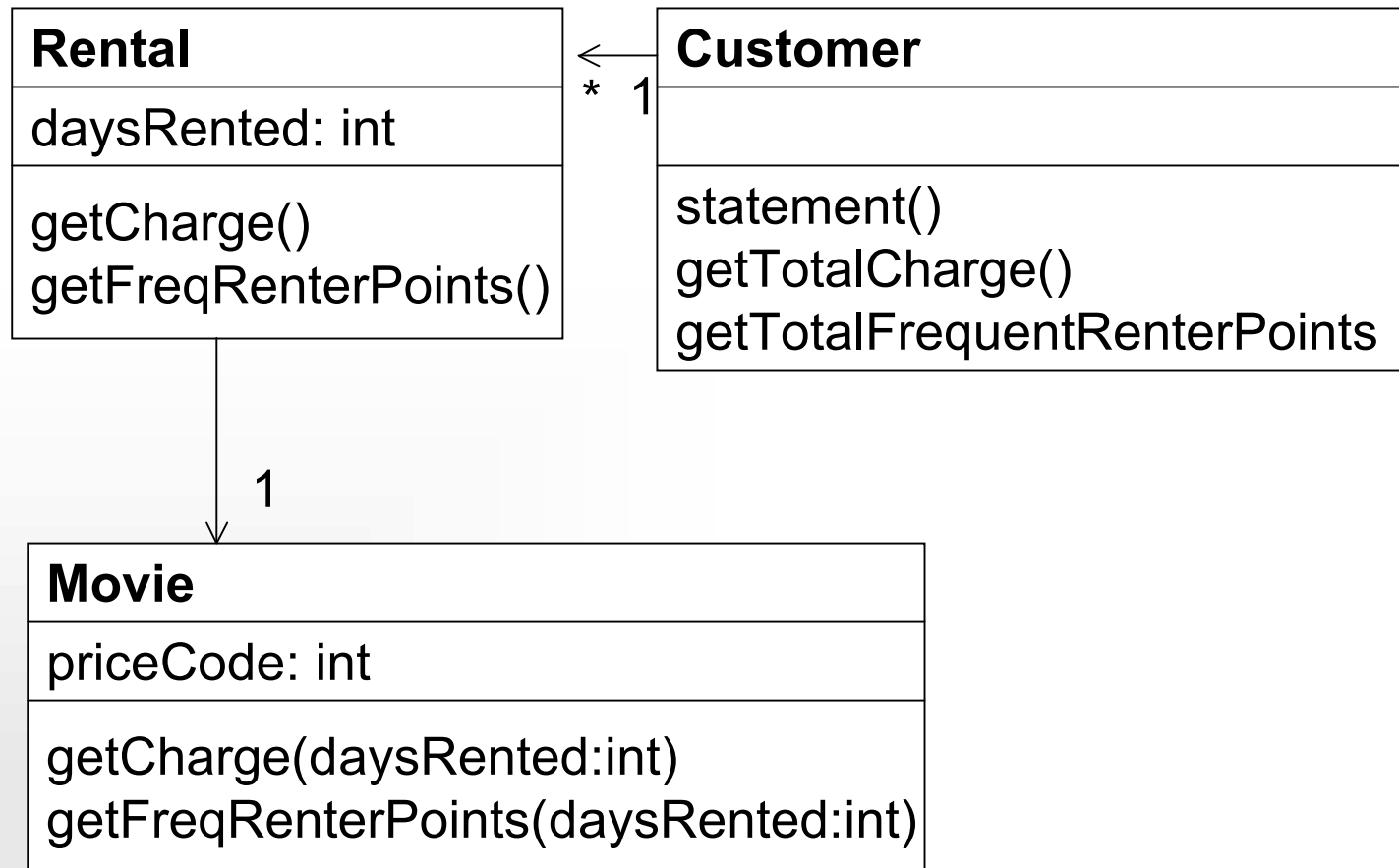
```
// class Rental...  
public double getCharge() {  
    return _movie.getCharge(_daysRented);  
}
```

- now I am passing information from **Rental** to **Movie** rather than from **Movie** to **Rental** as before
- why is this solution preferable over the previous one?

Because types have the potential to change...

7th Improvement

Moving frequentRenterPoints

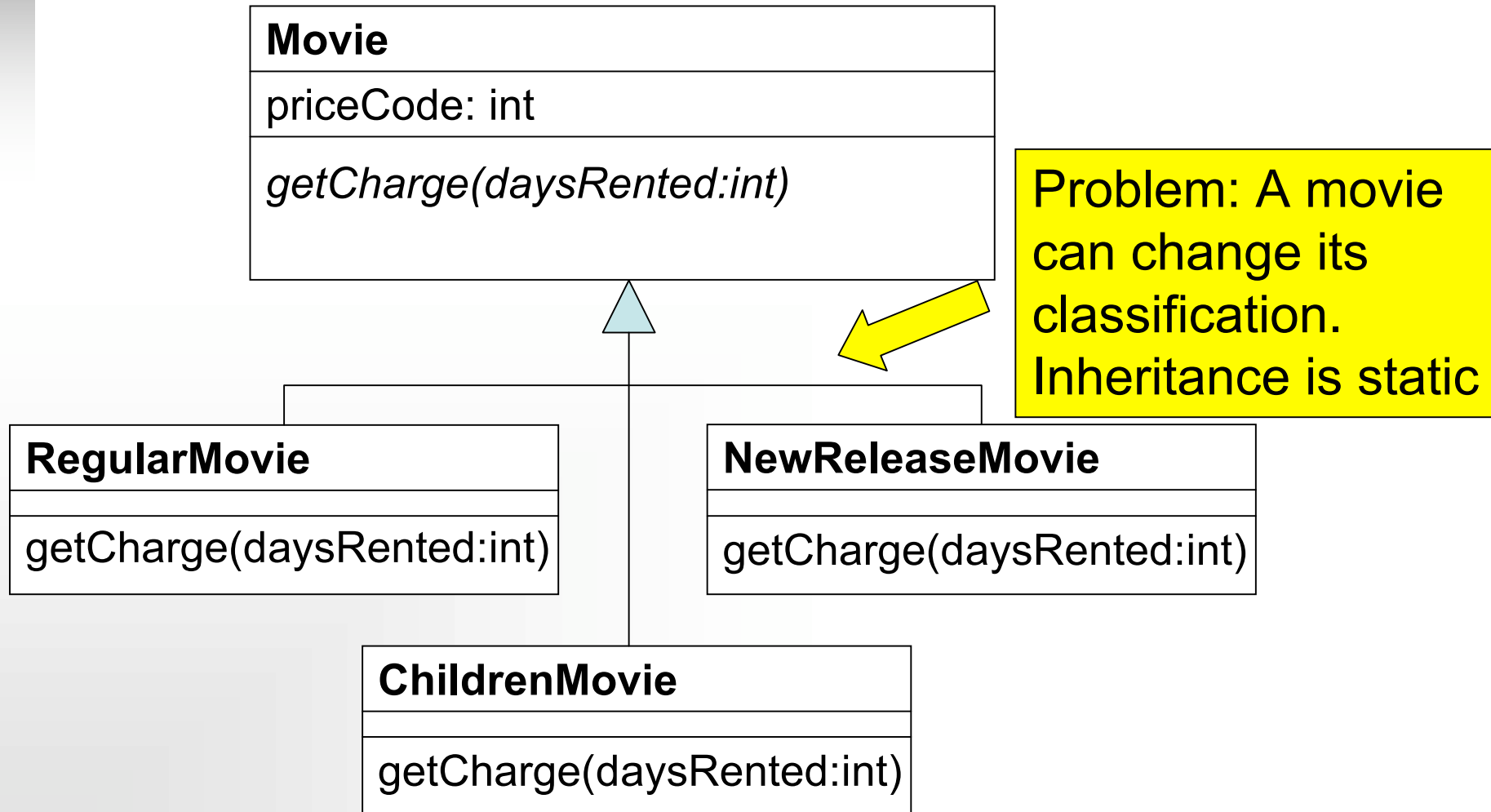


8th Improvement...?

- separate the definition of different movie types via inheritance!
- we have several types of movies which differ on the way they calculate the charge
- replace switch by polymorphism

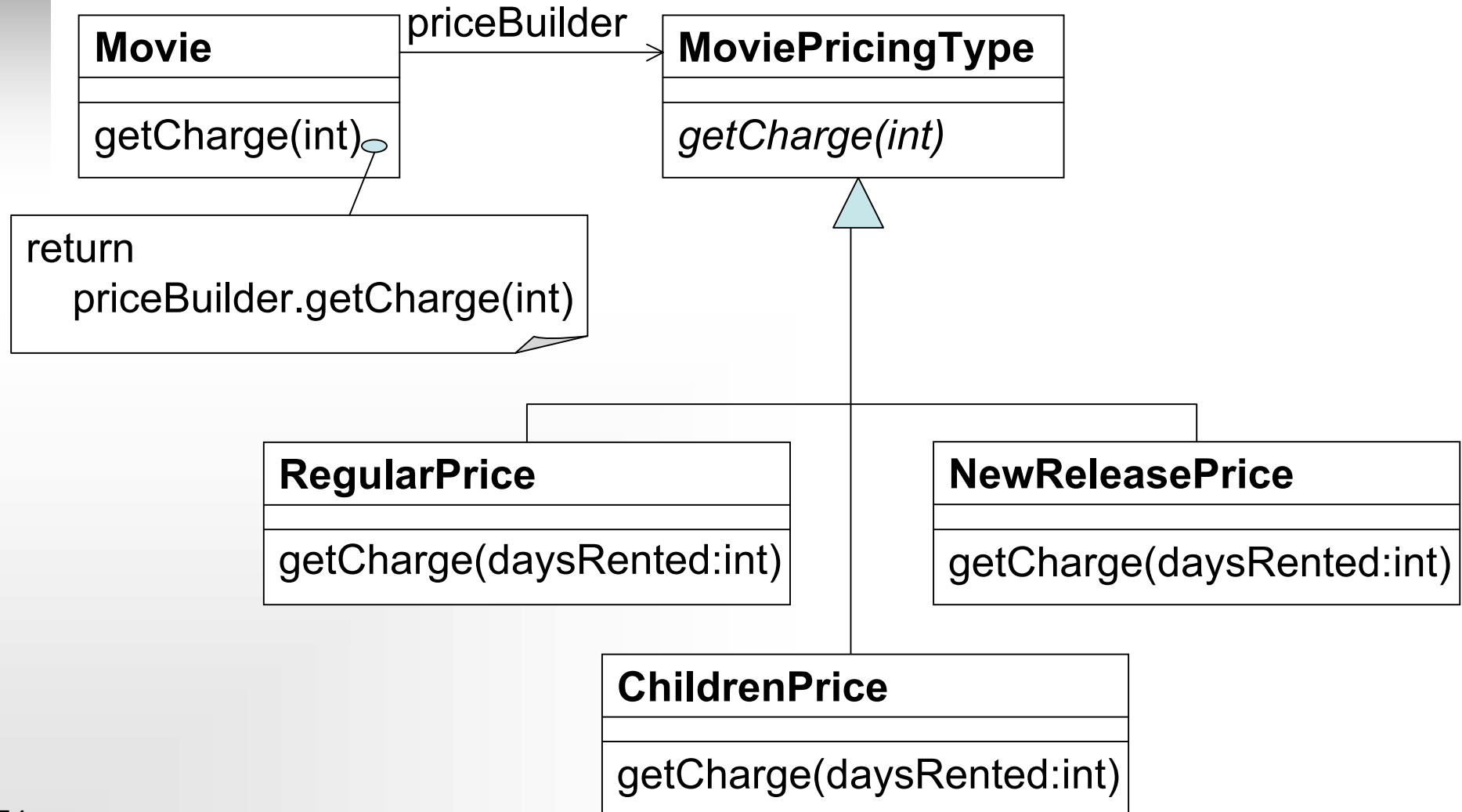
8th Improvement

Use Inheritance



8th Improvement

Use State/Strategy Pattern



8th Improvement

Use State/Strategy Pattern

- three steps to systematically introduce the strategy pattern
- correspond to three refactorings
 - *Replace Type Code with State/Strategy*
 - use *Move Method* to move the switch statement into the strategy class
 - *Replace Conditional with Polymorphism*

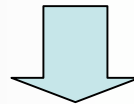
8th Improvement

Replace Type Code with State/Strategy

Step 1: *Self Encapsulated Field*

Ensure that all uses of the type code go through set and get methods.

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



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

Self Encapsulate Field

- You are accessing a field directly, but the coupling to the field is becoming awkward.
- Create getting and setting methods for the field and use only those to access the field.

```
private int _low, _high;  
boolean includes (int arg) {  
    return arg >= _low && arg <= _high;  
}
```



```
private int _low, _high;  
boolean includes (int arg) {  
    return arg >= getLow() && arg <= getHigh();  
}  
int getLow() {return _low;}  
int getHigh() {return _high;}
```

8th Improvement

Replace Type Code with State/Strategy

Step 2: *Add pricing classes*

```
class MoviePricingType {  
    abstract int getPriceCode();  
}
```

```
class ChildrenPricing  
extends MoviePricingType {  
    int getPriceCode() {  
        return Movie.CHILDRENS;  
    }  
}
```

```
class RegularPricing  
extends MoviePricingType {  
    int getPriceCode() {  
        return Movie.REGULAR;  
    }  
}
```

```
class NewReleasePricing  
extends MoviePricingType {  
    int getPriceCode() {  
        return Movie.NEW_RELEASE;  
    }  
}
```

8th Improvement

Replace Type Code with State/Strategy

Step 3: *Change movie's accessors for the price code to use the new classes*

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

8th Improvement

Replace Type Code with State/Strategy

Step 3: *Change movie's accessors for price code to use new classes*

```
class Movie {
    public int getPriceCode() {
        return _priceBuilder.getPriceCode();
    }
    public void setPriceCode(int arg){
        switch (arg) {
            case REGULAR:
                _priceBuilder = new RegularPricing();
                break;
            case CHILDRENS:
                _priceBuilder = new ChidrensPricing();
                break;
            case NEW_RELEASE;
                _priceBuilder = new NewReleasePricing();
                break:
        }
    }
    private MovingPricingType _priceBuilder;
}
```

8th Improvement

Replace Type Code with State/Strategy

Step 4: *Move conditional logic from Movie to MoviePricingType*

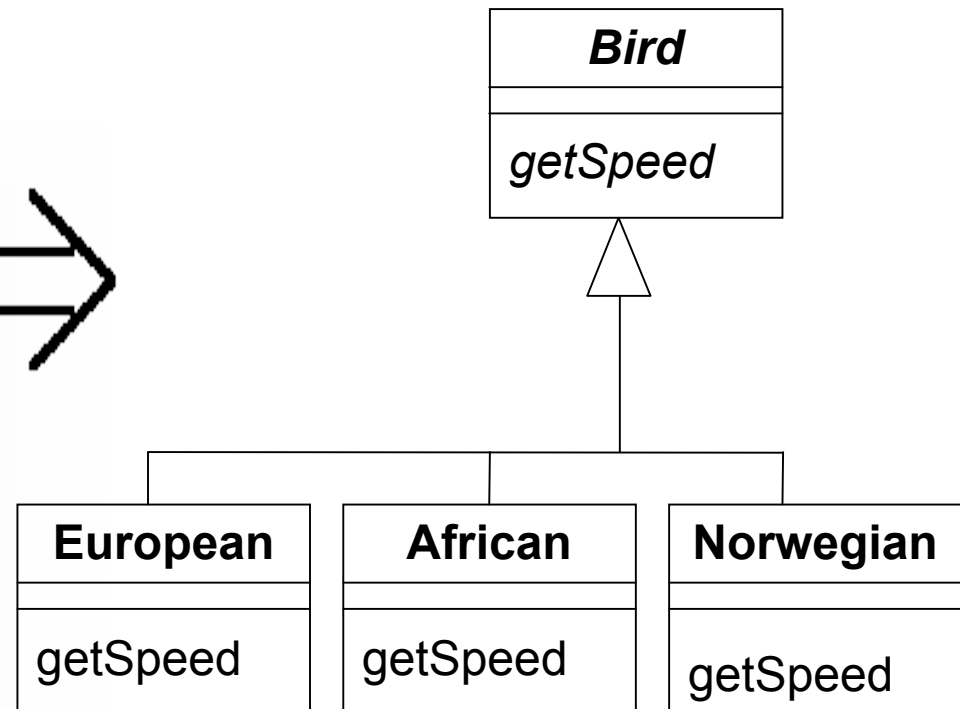
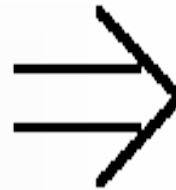
```
// class MoviePricingType...
public double getCharge(int daysRented) {
    double result = 0;
    switch(getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if(daysRented > 2) result += (daysRented - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
    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

```
// class Movie...
public double getCharge(int days) {
    return _priceBuilder.getCharge(days);
}
```

Repl. Conditional w/ Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object.
- Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

```
double getSpeed() {  
    switch (_type) {  
    case EUROPEAN:  
        return getBaseSpeed();  
    case AFRICAN:  
        return getBaseSpeed() -  
            getLoadFactor() *  
            _numberOfCoconuts;  
    case NORWEGIAN:  
        return (_isNailed) ? 0 :  
            getBaseSpeed(_voltage);  
    }  
}
```



8th Improvement

Replace Type Code with State/Strategy

Step 5: *Replace conditional logic with polymorphism*

```
// class MoviePricingType...
public 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;
}
```

8th Improvement

Replace Type Code with State/Strategy

Step 5: *Replace conditional logic with polymorphism*

```
// class RegularPricing...
public double getCharge(int daysRented) {
    double result = 2;
    if (daysRented > 2) result += (daysRented - 2) * 1.5;
    return result ;
}
```

```
// class ChildrensPricing...
public double getCharge(int daysRented) {
    double result = 1.5;
    if daysRented > 3) result += daysRented - 3) * 1.5;
    return result;
}
```

```
// class NewReleasePricing...
public double getCharge(int daysRented) {
    returns daysRented * 3;
}
```

8th Improvement

Replace Type Code with State/Strategy

Step 6: *Declare `getCharge()` abstract in `MoviePricingType`*

```
// class MoviePricingType...  
public abstract double getCharge(int daysRented);  
...
```

Refactoring in Customer

- when implementing `htmlStatement()` in `Customer`, I still did some copy-and-paste from the ASCII version
- so, I got some code duplication and even worse would need to do the same thing when adding supporting new formats
- I can further improve by factoring out methods for header and footer

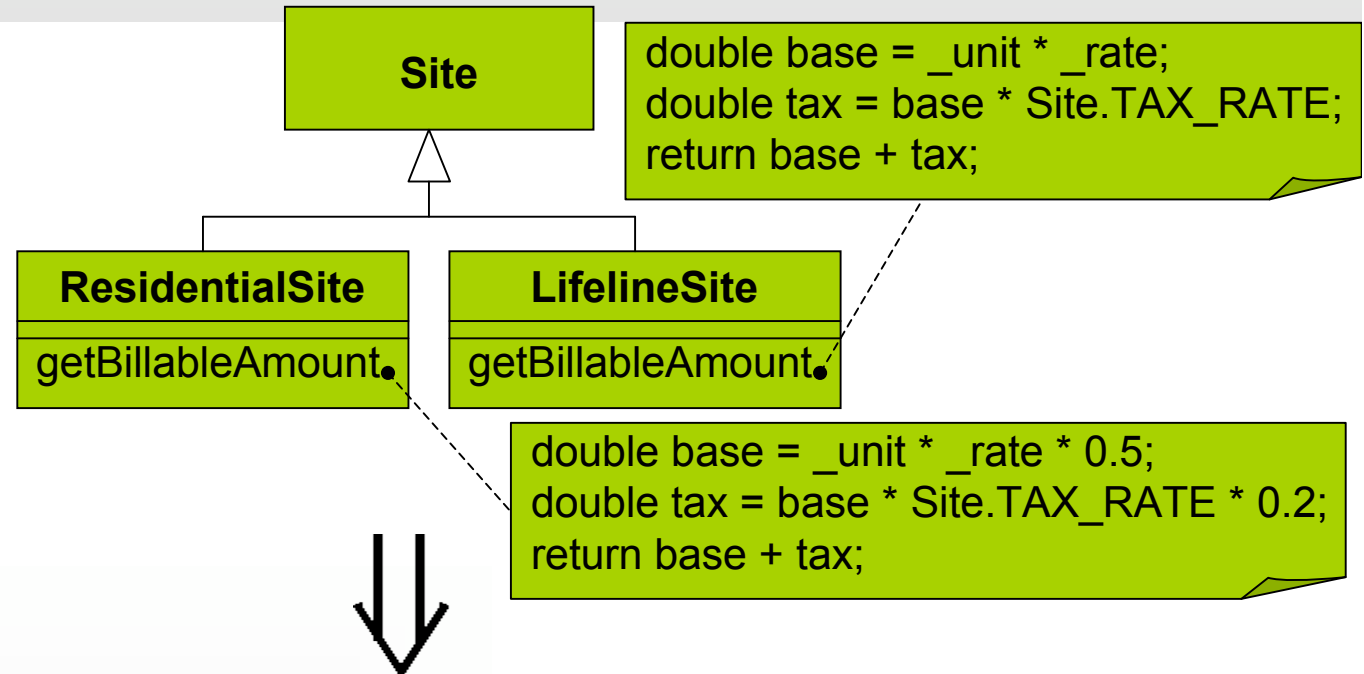
Any idea how?

Refactoring in Customer

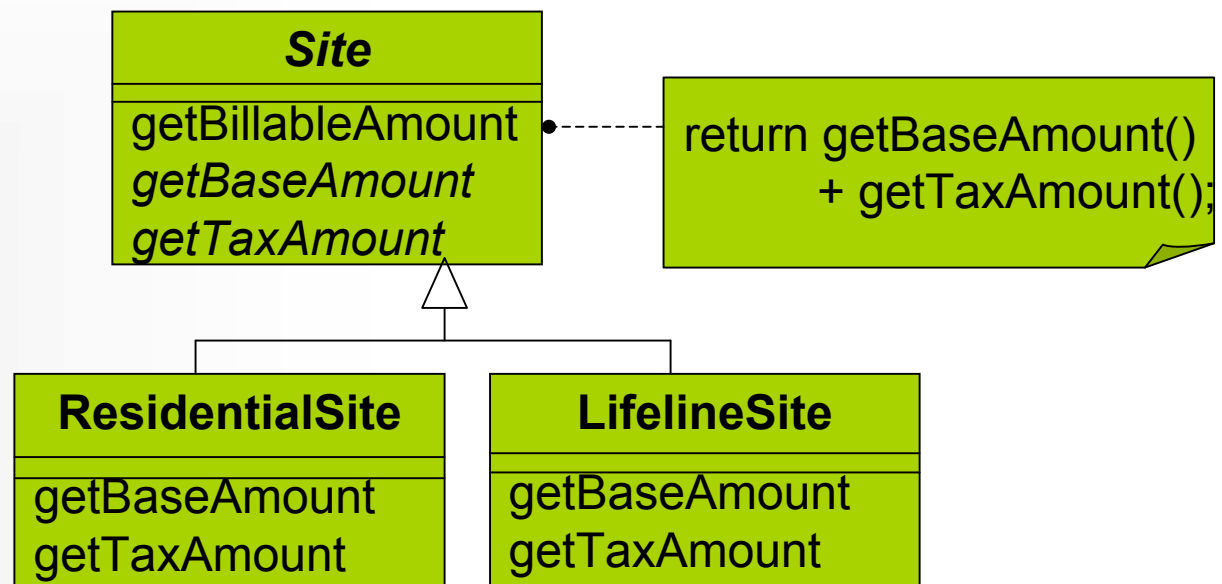
- use Strategy pattern instead of different methods for different formats
- use **Form Template Method** to define a generic control flow for generating statements of all kinds – independent of parts of the generation process that are dependent on the format

Form Template Method

You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.



Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.



Refactoring Defined

- **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 behaviour
- **Refactor** (verb): to restructure a software by applying a series of refactorings without changing its observable behaviour

Refactoring Defined

- is refactoring *just* cleaning up code?
- yes and no
 - it is cleaning up code
 - it is more than just that: it provides a technique for cleaning up code in a more efficient and controlled manner
 - cleaning up code gets faster
 - you know which refactorings to use
 - you know how to use them to minimise bugs
 - you test at every possible opportunity

Making Software...

- popular metaphor "software *construction*" (Bertrand Meyer, 1997)
- construction (e.g., of a building):
 1. an architect draws blueprints
 2. contractors dig the foundation, build the superstructure, wire and plumb, apply finishing touches
 3. tenants move in and live happily ever after, calling building maintenance to fix problems

...but does this apply to software?

Making Software...

- "software construction" is more like *gardening*
 - you plant many things according to some plan
 - some thrive, others end up as compost
 - move plants due to requirements
 - split and prune, pull weeds
 - constantly monitor garden's health, adjust layout
- this holds for software especially in today's agile processes where change is embraced

Why Should You Refactor?

- two kinds of values of a program:
 - what they can do for you today
 - what they can do for you tomorrow
- you can't program long without realising that what the system does today is only a part of the story
- I know enough to do the work today (hopefully). I don't know enough to do tomorrow's. But if I only work for today, I won't be able to work for tomorrow at all
- refactoring helps with that:
 - when you find that yesterday's decision doesn't make sense today, you change the decision. Now you can do today's work
 - and... you repeat for tomorrow's work

Why Should You Refactor?

- to improve the design of software
- to make software easier to understand
- to make it easier to find bugs
- to program faster

When Should You Refactor?

- refactoring is not an activity you set aside time to do
- refactoring is something you do all the time in little bursts
- you refactor because you want to do something else and refactoring helps you to do this other thing

*The first time you do something, you just do it.
The second time you do something similar,
you wince at the duplication, but you do the
duplication anyway. The third time you do
something similar, you refactor.*

Don Roberts

When Should You Refactor?

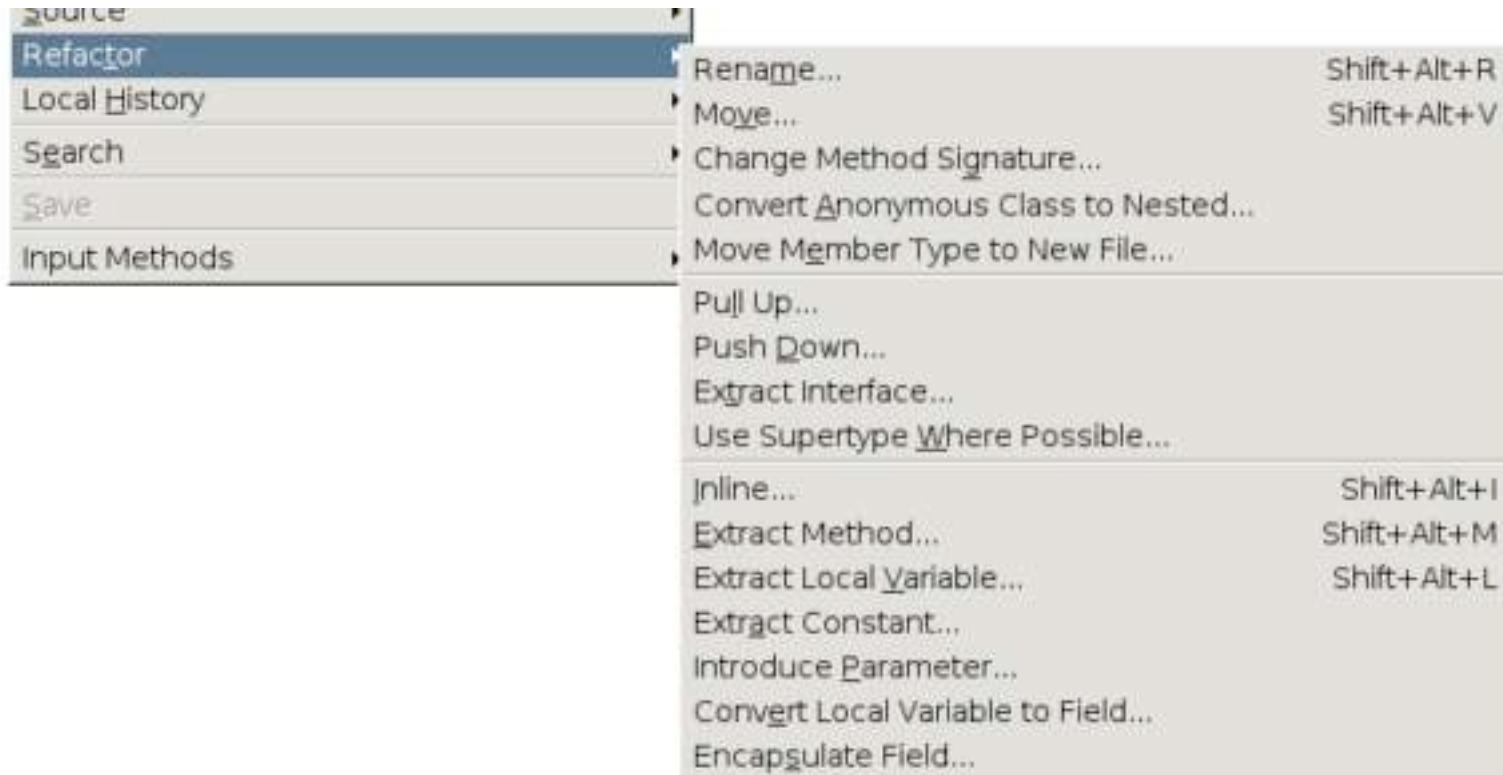
- when you...
 - add a function
 - need to fix a bug
 - do code reviewing

 - notice a duplication of knowledge
 - discover nonorthogonal (non-modular) design
 - want to update knowledge

- refactor early, refactor often

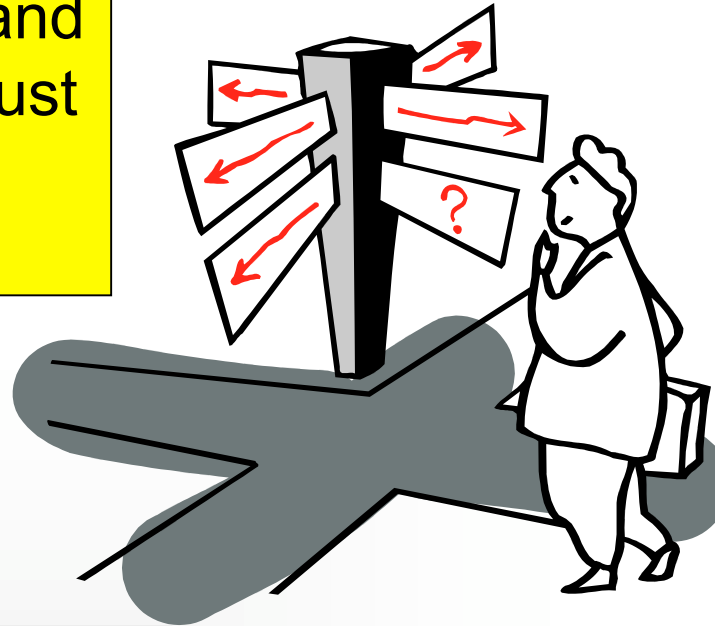
How to Refactor?

- simple tips (Martin Fowler)
 1. don't refactor and add functionality at the same time
 2. have good tests before you begin refactoring
 3. keep your steps small, test after each step
- use tools: refactoring does not have to be done manually



Refactoring and Design

Upfront design is the key piece and programming just mechanics



Refactoring is an alternative to design: you don't make any upfront design

Design changes are expensive: build flexibility into software wherever you can

Flexibility costs. Building flexibility in all places makes the overall system more complex. All the flexibility might not be needed anyway

Refactoring and Design

Refactoring can lead to simpler designs without sacrificing flexibility.

With refactoring you approach the risks of change differently.



You still think about potential changes, you still consider flexible solutions.

But, you don't have to think for tomorrow's flexibility.

Instead of implementing flexible solutions, you consider how difficult it is to get to them in the future via refactorings.

The other way around, you need to have a good sense about design to perform good refactorings.