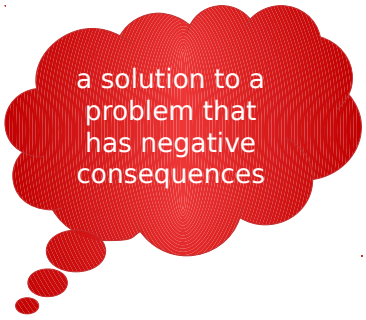
Abram Hindle
Department of Computing Science
University of Alberta

Refactoring

Slides originally by Ken Wong

Pattern



Anti-pattern

easier to recognize what is wrong (and try to fix it), than to get it "right" in the first place

Examples

Spaghetti code:
code with very complex, tangled control flow typified by lots of gotos

Dead code:
code whose "result" is no longer used

Refactoring

Idea:

change a software system so that the external behavior does not change but the internal structure is *improved*

do this in small steps (change a bit and re-test)

when adding a feature, refactor to make the addition easier to achieve

Code Smells

Bad Smells (in Code)

Quote:

"If it stinks, change it."
— Grandma Beck on child rearing

Exercise

Question:

What are some indicators or examples of poorly written code?

Bad Smells in Code

Goal:
critiquing code and software designs

Suggested indicators:
potential problems if left untouched

solutions require judgment and balance

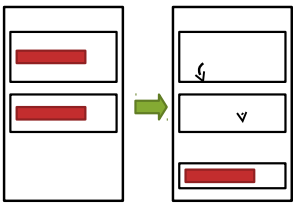
Duplicated Code

Indicator:
the same functionality appearing in more than one place

- e.g., same code in two methods of the same class
- e.g., same code in two sibling subclasses

Refactorings:
Extract Method

Pull Up Method



Long Method

Indicator:
long, difficult-to-understand methods

Why:
desire “short”, well-named methods

cohesive units of code

write a separate method instead of a comment

Refactoring:
Extract Method

Large Class (Blob or God Class)

Indicator:
a class trying to do too many things

- e.g., too many diverse instance variables

Why:
poor separation of concerns

Refactoring:
Extract Class

Divergent Change

Indicator:
when a class is commonly changed in different ways for different reasons

Why:
poor separation of concerns

Refactoring:
Extract Class

Shotgun Surgery

Indicator:
making a change requires many little changes across many different classes or methods

Why:
could miss a change

should consolidate these changes

Refactorings:
Move Method

Long Parameter List

Indicator:
passing in lots of parameters to a method (because "globals are bad")

Why:
difficult to understand

Refactorings:
Replace Parameter with Method

Introduce Parameter Object

Feature Envy

Indicator:
a method seems more interested in the details of a class other than the one it is in

- e.g., invoking lots of get methods on another class

```
int length = rect.getLength();
int width = rect.getWidth();
int area = length * width;
```

Why:
this behavior may belong in the other class

Refactorings:
Move Method

Extract Method

```
int area = rect.area();
```

Data Class

Indicator:
classes that are just data (manipulated by other classes with getters and setters)

Refactorings:
Encapsulate Field

Extract Method

Move Method

Data Clumps

Indicator:
groups of data appearing together in the instance variables of classes, parameters to methods, etc.

Refactorings:
Extract Class

Introduce Parameter Object

```
public void doSomething( int x, int y, int z ) {
...
}
```


Primitive Obsession

Indicator:
using the built-in types too much

- e.g., “stringitus”, everything being a string

```
public static void checkCode( String postalCode ) {
...
}
```

Why:
leads to non-object-oriented designs

Refactoring:
Replace Data Value with Object

Switch Statements

Indicator:
long conditionals on type codes defined in other classes

Refactorings:
Extract Method, Move Method

Replace Type Code

Replace Conditional with Polymorphism

Speculative Generality

Indicator:
"we might need this someday"

- e.g., an unused abstraction, hook, or parameter

Why:
increases design complexity

Refactorings:
Collapse Hierarchy

Remove Parameter

Message Chains

Indicator:
long chains of navigation to get to an object

Why:
`a.getB().getC().doSomething();`
poor flexibility and testability

could be Law of Demeter violation

Refactoring:
Hide Delegate

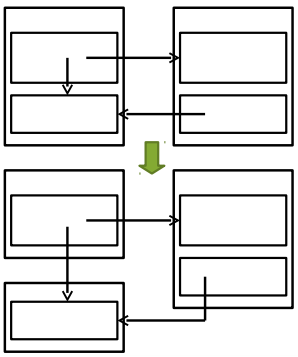
Inappropriate Intimacy

Indicator:
two classes that depend too much on each other, with lots of bidirectional communication

Why:
high coupling

Refactorings:
Move Method

Extract Class



Refused Bequest

Indicator:
when a subclass inherits something that is not needed

when a superclass does not contain truly common state or behavior

Refactorings:
Push Down Method and Push Down Field

Replace Inheritance with Delegation

Comments

Why:

could be “deodorant” for bad smelling code

simplify and refactor so comment is not needed

use comments to explain *why* something was done a certain way

Refactorings:

Extract Method

Rename Method

Refactoring Example

Refactoring Example

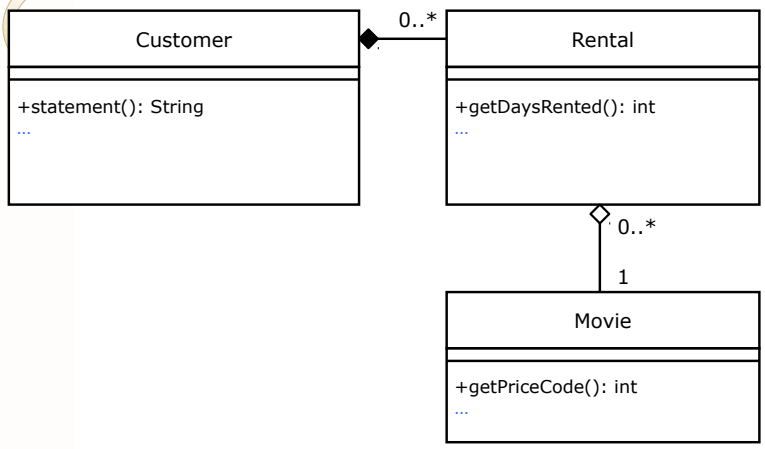
Problem:

a program to calculate and print a statement of a customer’s charges at a video store:

- customer can rent movies
- movies have different pricing
- movies are rented for a number of days
- customer can collect frequent renter points

what kind of design?

Initial Structural Design



```
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;
    }
}
```



```
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 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;
            }
            …
```

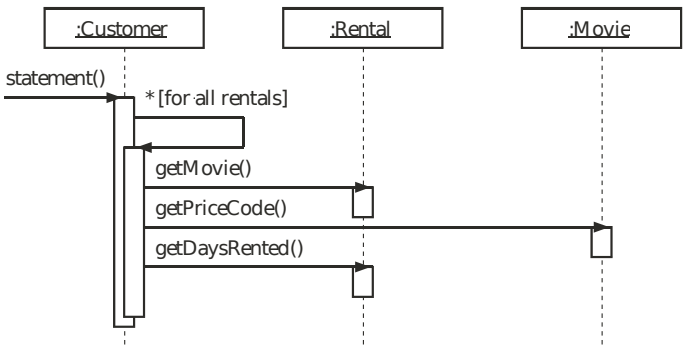


```
            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1)
                frequentRenterPoints++;

            // 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;
    }
}
```


Initial Behavioral Design



Code Smells

What smells?

Code Smells

Issues:

procedural, not object-oriented programming

`statement()` method does too much

Customer class is a blob class

potentially difficult to add features

- e.g., HTML output
- e.g., new charging rules

Refactoring

Idea:

if the code is not structured conveniently to add a feature, first *refactor* the program to make it easy to add the feature, then add the feature

small steps

Refactoring

First step:

need unit tests

Extract Method

Goal:

decompose `statement()` method

extract logical chunk of code as a new method

Extract Method (Before)

```
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;
                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;
            }
            …
```


Extract Method (After)

```
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();

            thisAmount = amountFor( each );

            …
```


Extract Method (After)

```
public class Customer {
    …
    private 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;
    }
    …
}
```

Extract Method

Compile and test!

small steps

Rename Variables

Goal:

rename variables in `amountFor()`

enhance readability

Rename Variables (Before)

```
public class Customer {
    …
    private 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;
    }
    …
}
```


Rename Variables (After)

```
public 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;
    }
    …
}
```

Rename Variables

Compile and test.

Anything unusual?

Move Method

Refactoring:

move `amountFor()` to `Rental` class

- method uses *rental* information, but not *customer* information

move this method to the right class

Move Method (Before)

```
public 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;
    }
    …
}
```


Move Method (After)

```
public 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;
    }
    …
}
```

Move Method (After)

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


Move Method

Compile and test.

Cleanup indirection ...

Move Method (Continued)

Refactoring:

replace references to `amountFor()` with `getCharge()`

adjust references to old method to use new method

remove old method

Move Method (Continued)

```
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();

            thisAmount = amountFor( each );

            …
```

Move Method (Continued)

```
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();

            thisAmount = each.getCharge();

            …
```


Move Method (Continued)

Compile and test.

Replace Temp with Query

Refactoring:

eliminate `thisAmount` temporary in `statement()`

replace redundant temporary variable with query

Replace Temp (Before)

```
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();

            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;
        }
        …
```

Replace Temp (After)

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


Extract/Move Method

Refactoring:

similarly, extract frequent renter points logic

- applicable rules go with the rental, not customer

Extract/Move Method (Before)

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


Extract/Move Method (After)

```
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()) {
            Rental each = (Rental)rentals.nextElement();

            // add frequent renter points
            frequentRenterPoints += each.getFrequentRenterPoints();

            // show figures for this rental
            result += “\t” + each.getMovie().getTitle() + “\t” +
                String.valueOf( each.getCharge() ) + “\n”;
            totalAmount += each.getCharge();
        }
        …
```

Extract/Move Method (After)

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


Replace Temp with Query

Refactoring:

eliminate `totalAmount` temporary and replace with `getTotalCharge()` query

Replace Temp w/ Query (Before)

```
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()) {
            Rental each = (Rental)rentals.nextElement();

            // add frequent renter points
            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;
    }
}
```


Replace Temp w/ Query (After)

```
public 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();

            // add frequent renter points
            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;
    }
}
```

Replace Temp w/ Query (After)

```
public 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;
    }
    …
}
```


Replace Temp with Query

Refactoring:

eliminate `frequentRenterPoints` temporary and replace with `getTotalFrequentRenterPoints()` query

Replace Temp w/ Query (Before)

```
public 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();

            // add frequent renter points
            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;
    }
}
```

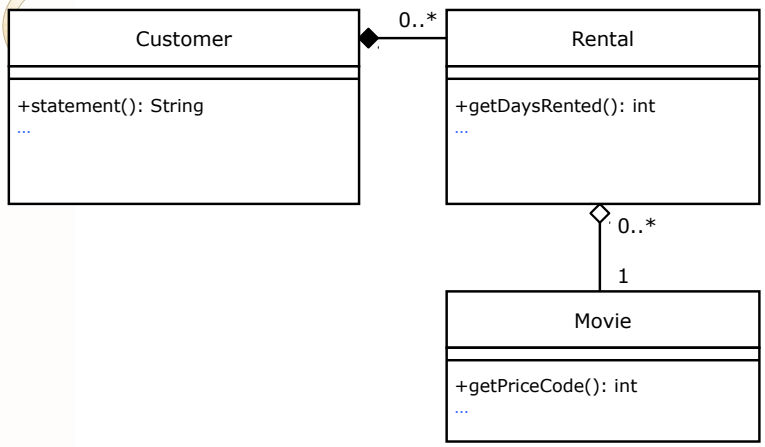

Replace Temp w/ Query (After)

```
public 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 += “\t” + each.getMovie().getTitle() + “\t” +
                String.valueOf( each.getCharge() ) + “\n”;
        }

        // add footer lines
        result += “Amount owed is “ +
            String.valueOf( getTotalCharge() ) + “\n”;
        result += “You earned “ +
            String.valueOf( getTotalFrequentRenterPoints() ) +
            “ frequent renter points”;
        return result;
    }
}
```

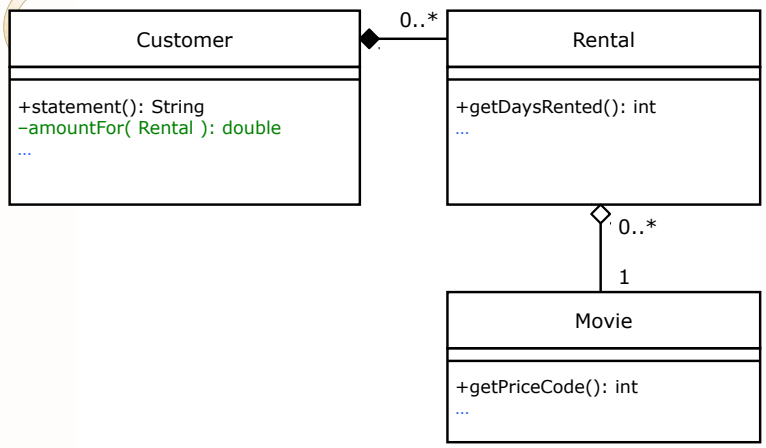
Replace Temp w/ Query (After)

```
public class Customer {
    …
    private int getTotalFrequentRenterPoints() {
        int result = 0;
        Enumeration rentals = _rentals.elements();

        while (rentals.hasMoreElements()) {
            Rental each = (Rental)rentals.nextElement();

            result += each.getFrequentRenterPoints();
        }
        return result;
    }
    …
}
```

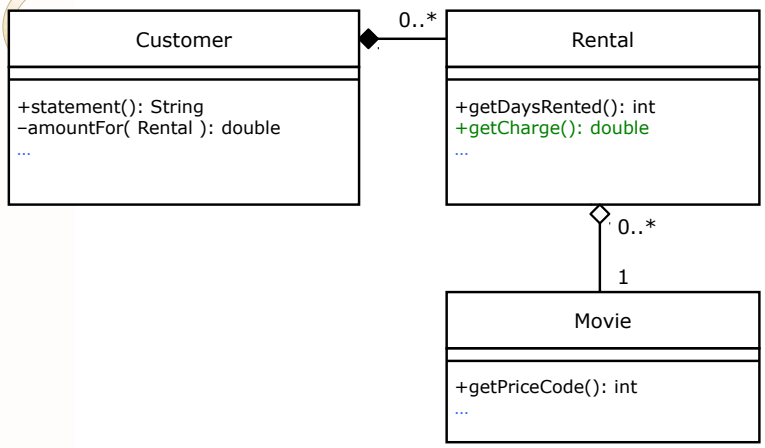

Initial Structural Design



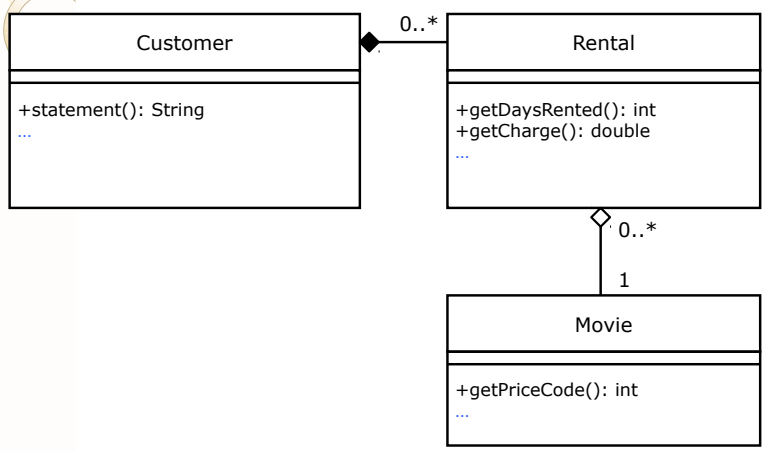
After Extract Method



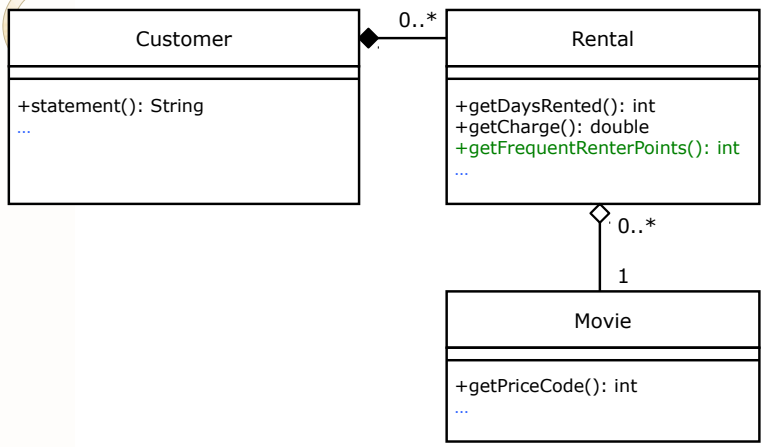
During Move Method



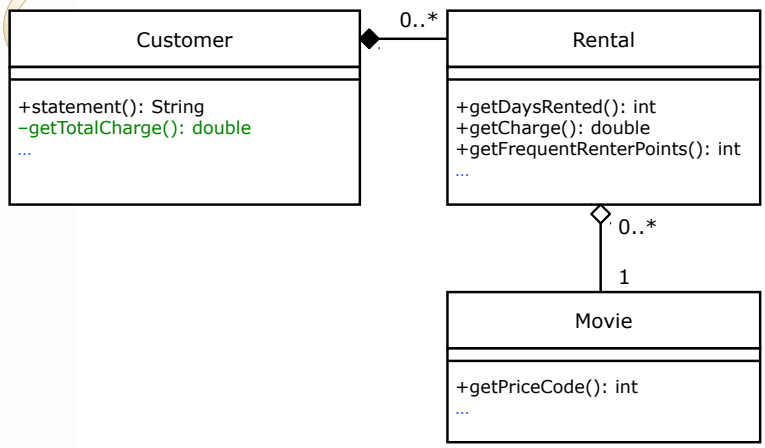
After Move Method



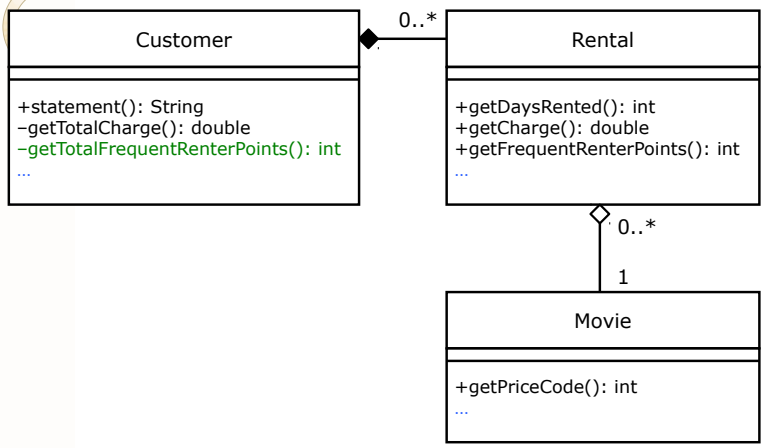
After Extract/Move Method



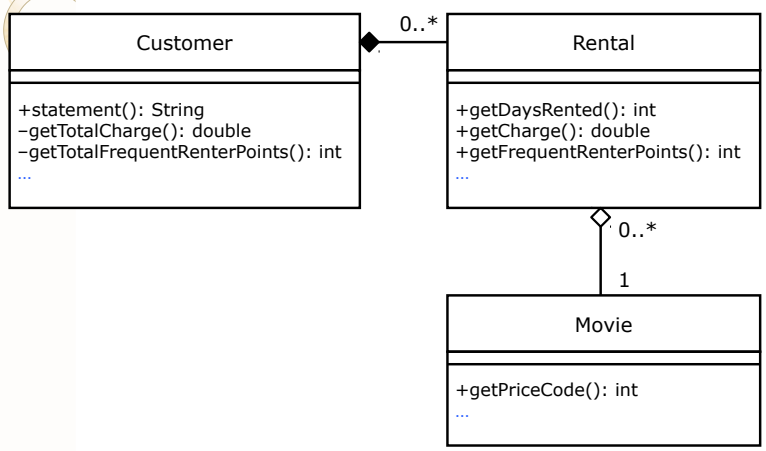
After Replace Temp w/ Query



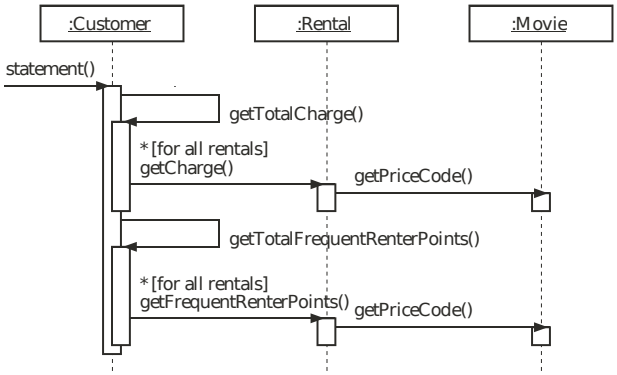
After Replace Temp w/ Query



Second Structural Design



Second Behavioral Design



Refactoring

Consequences:

more object-oriented

- decomposes big methods into smaller ones
- distributes responsibilities among classes

more code

slower performance?

New HTML Output Feature

```
public class Customer {
    …
    public String htmlStatement() {
        Enumeration rentals = _rentals.elements();
        String result = “<h1>Rental Record for “ + getName() + “</h1>\n”;

        while (rentals.hasMoreElements()) {
            Rental each = (Rental)rentals.nextElement();

            // show figures for this rental
            result += each.getMovie().getTitle() + “: “ +
                String.valueOf( each.getCharge() ) + “<br>\n”;
        }

        // add footer lines
        result += “<p>Amount owed is “ +
            String.valueOf( getTotalCharge() ) + “</p>\n”;
        result += “<p>You earned “ +
            String.valueOf( getTotalFrequentRenterPoints() ) +
            “ frequent renter points</p>”;
        return result;
    }
}
```

Changing Needs

Feature:

new price classifications of movies

Move Method

Refactoring:

rental logic should not depend on *specific* movie types

Move Method (Before)

```
public 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;
    }
    …
}
```


Move Method (After)

```
public 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;
    }
    …
}
```

Move Method (After)

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


Move Method (Before)

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


Move Method (After)

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

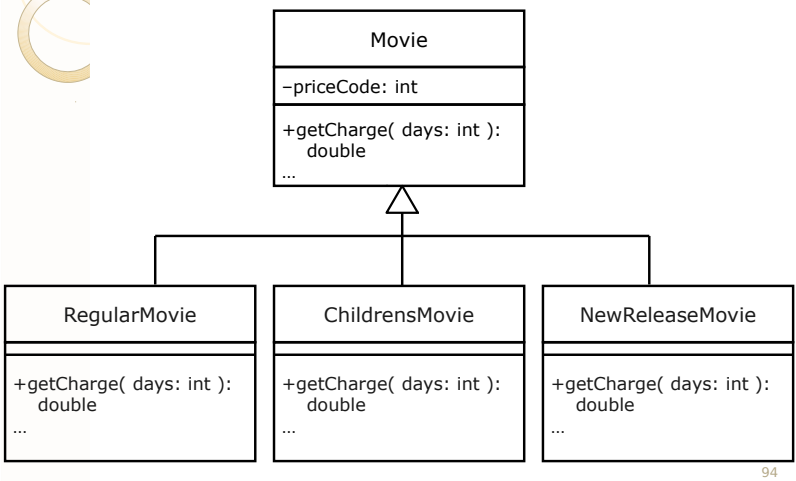

Move Method (After)

```
public class Rental {
    …
    public int getFrequentRenterPoints() {
        return _movie.getFrequentRenterPoints( _daysRented );
    }
    …
}
```

Replace Conditional Logic

Ready for inheritance? ...

Proposed Redesign?



Proposed Redesign

Flaw:

- a movie may change its classification during its lifetime (e.g., new release to regular)
- but, an object cannot change its class during its lifetime

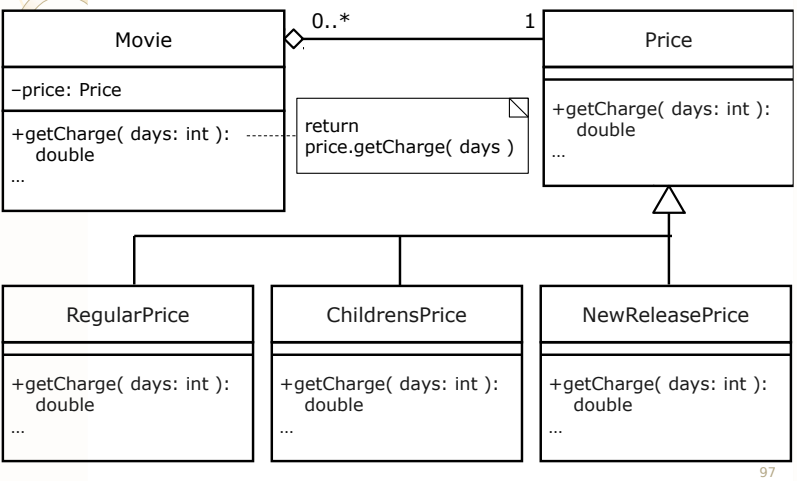
solution?

Replace Conditional Logic

Idea:

- use Price (state) objects
- State design pattern

Replace Conditional Logic



Replace Type Code with State

Refactoring:

replace price (type) code

compile and test after each step

first, make sure uses of the price type code go through accessor methods ...

Replace Type Code with State

```
public class Movie {
    …
    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;
    }
    …
}
```


Replace Type Code with State

```
public class Movie {
    …
    private int _priceCode;

    public Movie( String title, int priceCode ) {
        _title = title;
        setPriceCode( priceCode );
    }
    public int getPriceCode() {
        return _priceCode;
    }
    public void setPriceCode( int arg ) {
        _priceCode = arg;
    }
    …
}
```

Replace Type Code with State

Refactoring:

add new state classes ...

Replace Type Code with State

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

class RegularPrice extends Price {
    public int getPriceCode() {
        return Movie.REGULAR;
    }
}

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

class ChildrensPrice extends Price {
    public int getPriceCode() {
        return Movie.CHILDRENS;
    }
}
```


Replace Type Code with State

Refactoring:

replace price type codes with instances of price state classes ...

Replace Type Code with State

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

Replace Type Code with State

```
public class Movie {
    …
    private Price _price;
    …
    public int getPriceCode() {
        return _price.getPriceCode();
    }
    public void setPriceCode( int arg ) {
        switch (arg) {
            case REGULAR:
                _price = new RegularPrice();
                break;
            case NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
            case CHILDRENS:
                _price = new ChildrensPrice();
                break;
            default:
                throw new IllegalArgumentException(
                    “Incorrect price code” );
        }
    }
    …
}
```


Move Method

Refactoring:

move `getCharge()` to `Price` class

Move Method (Before)

```
public 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;
    }
    …
}
```


Move Method (After)

```
public class Price {
    …
    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;
    }
    …
}
```

Move Method (After)

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


Replace Conditional with Polymorphism

Refactoring:

replace switch statement in `getCharge()`

define abstract method

for each case, add overriding method

Replace Conditional with Polymorphism (Before)

```
class Price {
    …
    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;
    }
    …
}
```


Replace Conditional with Polymorphism (After)

```
class RegularPrice {
    public double getCharge( int daysRented ) {
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    }
}
class NewReleasePrice {
    public double getCharge( int daysRented ) {
        return daysRented * 3;
    }
}
class ChildrensPrice {
    public double getCharge( int daysRented ) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }
}
```

Replace Conditional with Polymorphism (After)

```
class Price {
    …
    public abstract double getCharge( int daysRented );
    …
}
```


Move Method

Refactoring:

move `getFrequentRenterPoints()` to `Price` class

Move Method (Before)

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


Move Method (After)

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

Move Method (After)

```
public class Movie {
    …
    public int getFrequentRenterPoints( int daysRented ) {
        return _price.getFrequentRenterPoints( daysRented );
    }
    …
}
```


Replace Conditional with Polymorphism

Refactoring:

replace if statement in `getFrequentRenterPoints()`

Replace Conditional with Polymorphism (Before)

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


Replace Conditional with Polymorphism (After)

```
class Price {
    …
    public int getFrequentRenterPoints( int daysRented ) {
        return 1;
    }
    …
}

class NewReleasePrice {
    …
    public int getFrequentRenterPoints( int daysRented ) {
        return (daysRented > 1) ? 2 : 1;
    }
    …
}
```

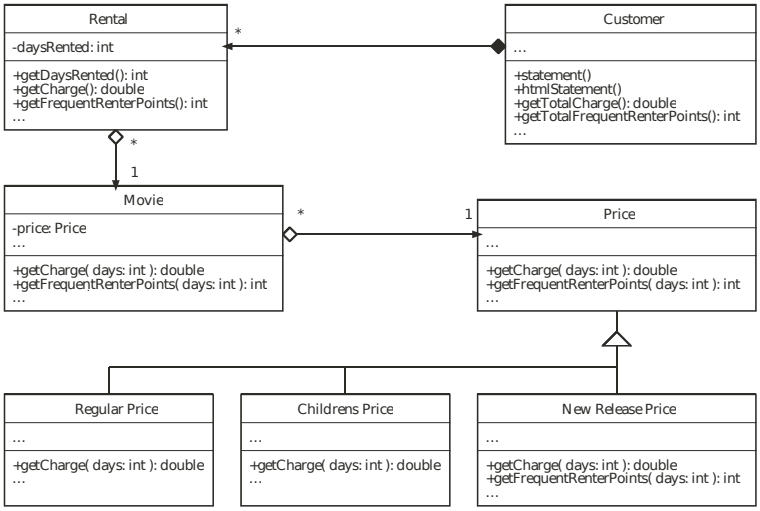
Refactoring

Result:

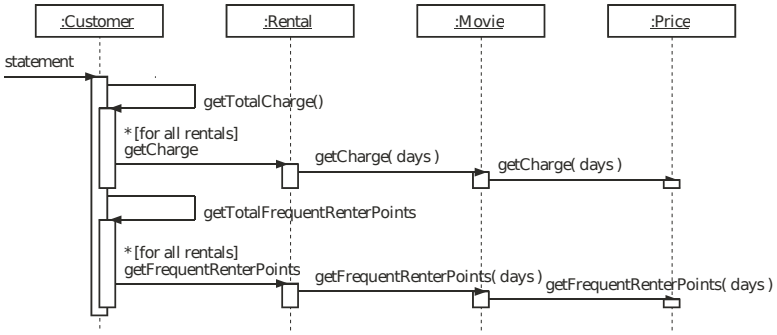
easier to change price behavior

- change movie classifications
- change rules for charging and frequent renter points

rest of application does not know about this use of the State design pattern



Third Behavioral Design



- **Refactoring Principles**

Refactoring

Basic principles:

- catalog of refactorings
- do not change outward behavior
- reduce risk of change
- one thing at a time
- test each step
- iterate

Refactoring

Outcomes:

- encode design intent within class structure
- reorganizing code
- sharing logic
- express conditional logic

Refactoring

Potential limitations:

- too much indirection
- performance impact
- changing published interfaces
- are significant design changes possible?

Refactoring

When not to refactor:

- when you should rewrite
- when you are close to a deadline

Refactoring

An analogy:

unfinished refactoring is like going into debt

debt is fine as long as you can meet the interest payments (extra maintenance costs)

if there is too much debt, you will be overwhelmed

—Ward Cunningham

Redesigning with Patterns

Common causes of change in client code:

creating an object by naming the class directly

- fix with Abstract Factory or Factory Method

dependence on specific hard-code requests

- fix with Chain of Responsibility or Command

algorithmic dependencies

- fix with Template Method

tight coupling

- fix with Façade or Observer

too much subclassing

- fix with Decorator

Kinds of Refactorings

Creating methods:

intended to help reduce the size of methods and improve the readability of the code

Extract Method,
Inline Method,
Replace Temp with Query

Kinds of Refactorings

Moving features between objects:

sometimes, responsibility is placed in the wrong class or a class ends up with too many responsibilities

Move Method,
Move Field,
Extract Class

Kinds of Refactorings

Organizing data:
sometimes, objects can be used instead of simple data items

Replace Data Value with Object,
Replace Array with Object

Kinds of Refactorings

Simplifying conditional expressions:
conditional expressions and logic can be difficult to understand

Replace Conditional with Polymorphism

Kinds of Refactorings

Making method calls simpler:
complicated programming interfaces can be difficult to use

Rename Method,
Add Parameter

Kinds of Refactorings

Dealing with generalization:
getting methods and subclasses to the right place

Pull Up Method,
Push Down Method,
Extract Subclass,
Extract Superclass

Java Practices

[Haggar, 2000]:

- *general techniques*
- objects and equality
- exception handling
- performance
- multithreading
- *classes and interfaces*

Java General Techniques

Understand that all non-static methods can be overridden by default

- using `final` prevents a subclass from overriding a method

Choose carefully between arrays and Vectors

- know their characteristics (element types, growable, speed)

Java General Techniques

Prefer polymorphism to `instanceof`

- many uses of `instanceof` can be eliminated with polymorphism, which creates more extensible code.

Use `instanceof` only when you must

- e.g., if you must safely downcast

Java General Techniques

Set object references to `null` when they are no longer needed
- even with garbage collection, still need to pay attention to memory usage

Java Classes and Interfaces

Define and implement immutable classes judiciously
- sometimes want objects that do not change
- e.g., a color object
- how?

Java Classes and Interfaces

Enabling immutability for a class:
- declare all data `private`
- set all data in the constructor
- only getter methods; no setter methods
- declare the class `final`
- clone mutable objects before returning a reference to them from a getter method
- clone objects provided to the constructor that are references to mutable objects

Java Classes and Interfaces

Use inheritance or delegation to define immutable classes from mutable ones
- reference a mutable object through an immutable interface
  - does not prevent casting the reference
- have an immutable object delegate to the mutable object
- have immutable abstract class and derived classes with mutable and immutable implementations

Effective Java

[Bloch 2001]:

- creating and destroying objects
- *methods common to all objects*
- *classes and interfaces*
- substitutes for C constructs
- *methods*
- general programming
- exceptions
- threads
- serialization

Effective Java

Methods common to all objects:

- obey the general contract when overriding `equals()`
- always override `hashCode()` when you override equals
- always override `toString()`
- override `clone()` judiciously
- consider implementing `Comparable`

Effective Java

Classes and interfaces:

- minimize the accessibility of classes and members
- favor composition over inheritance
- design and document for inheritance or else prohibit it
- prefer interfaces to abstract classes
- use interfaces only to define types

Effective Java

Methods:

- check parameters for validity
- make defensive copies when needed
- design method signatures carefully
- return zero-length arrays, not nulls
- write doc comments for all exposed API elements

More Information

Books:

Refactoring

- M. Fowler
- Addison-Wesley, 1999

AntiPatterns

- W.J. Brown, R. C. Malveau, H. W. McCormick III, T.J. Mowbray
- Wiley, 1998

More Information

Books:

Practical Java

- P. Haggar
- Addison-Wesley, 2000

Effective Java

- J. Bloch
- Addison-Wesley, 2001

More Information

Articles:

"Cloning Considered Harmful" Considered Harmful

- C. Kapser and M. W. Godfrey
- WCRE 2006 Proceedings, IEEE CS Press

More Information

Links:

Refactoring Home Page

- http://refactoring.com/

The 7 Deadly Sins of Software Development

- http://www.javaworld.com/javaworld/jw-02-2011/110217-fatal-exception.html