Refactoring

Abram Hindle
hindle1@ualberta.ca

Henry Tang
hktang@ualberta.ca

Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. 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 go-tos
- Dead code:
  - Code whose “result” is no longer used

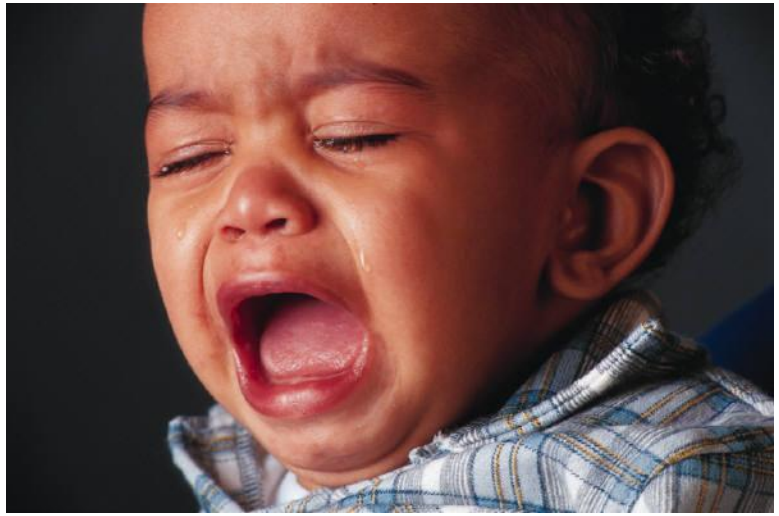
Refactoring

- 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)

- “If it stinks, change it.”
  — Grandma Beck on child rearing

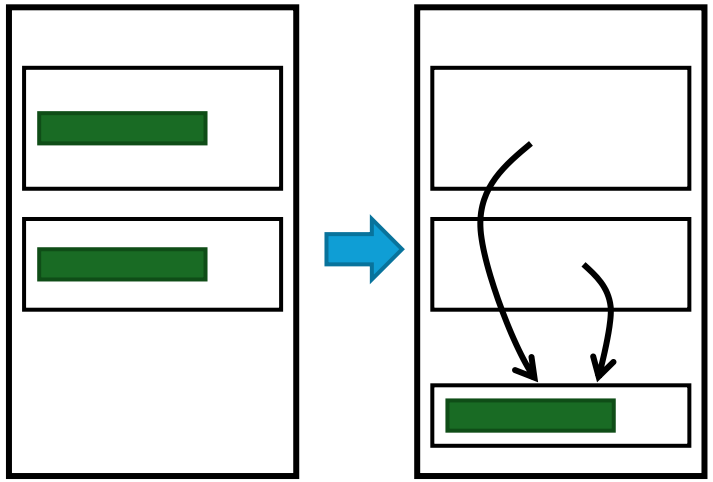


Bad Smells in Code

- 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
- Why:
  - This behavior may belong in the other class
- Refactorings:
  - Move Method
  - Extract Method

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

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

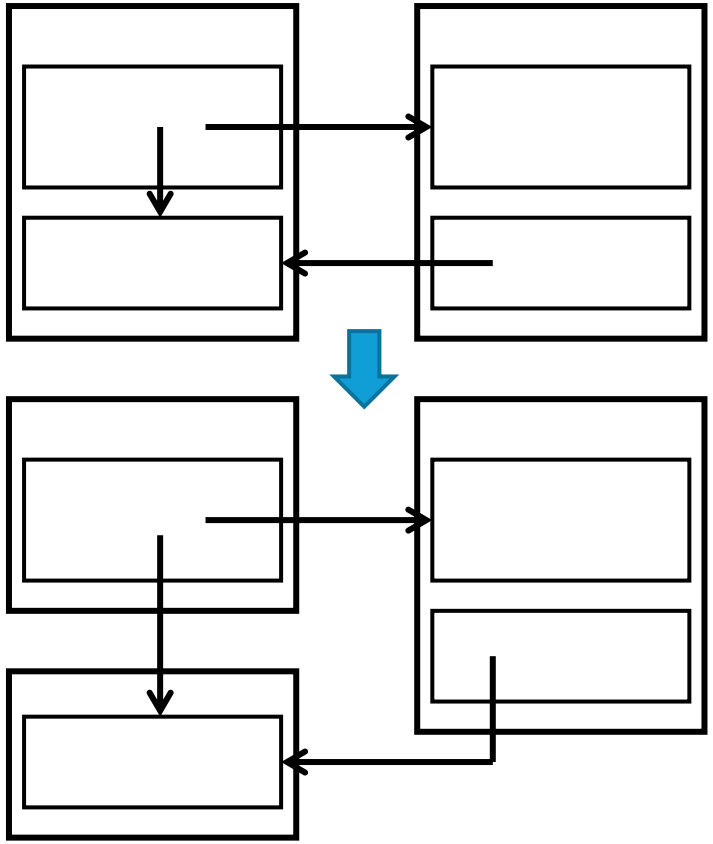
- Indicator:
  - Long chains of navigation to get to an object

```
a.getB().getC().doSomething();
```

- Why:
  - Poor flexibility and testability
- 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

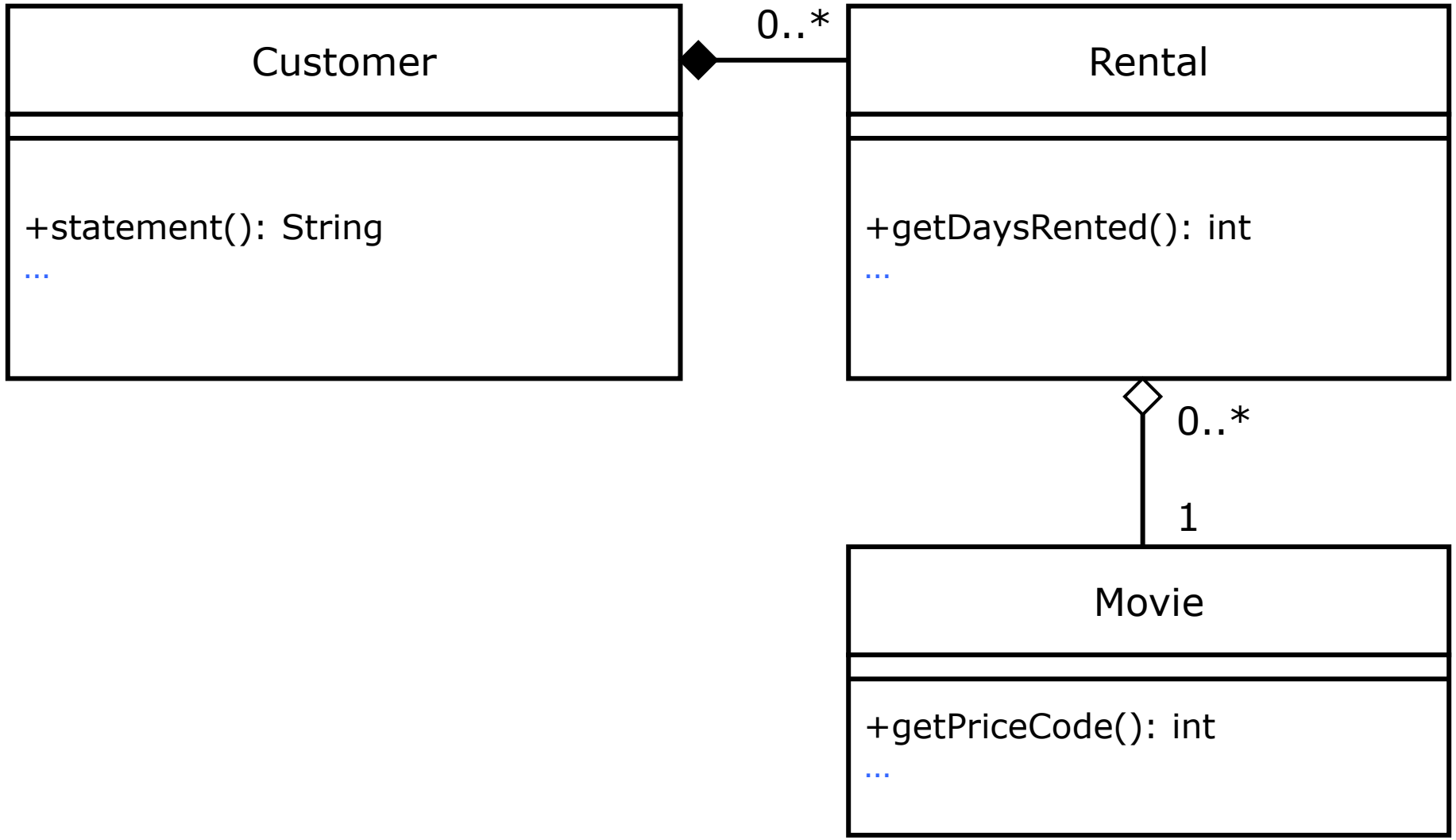
- Why:
  - Could be “deodorant” for bad smelling code
  - Simplify and refactor so comment is not needed
  - Ise comments to explain *why* something was done a certain way
- Refactorings:
  - Extract Method
  - Rename Method

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 several 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;
      }
      …
```

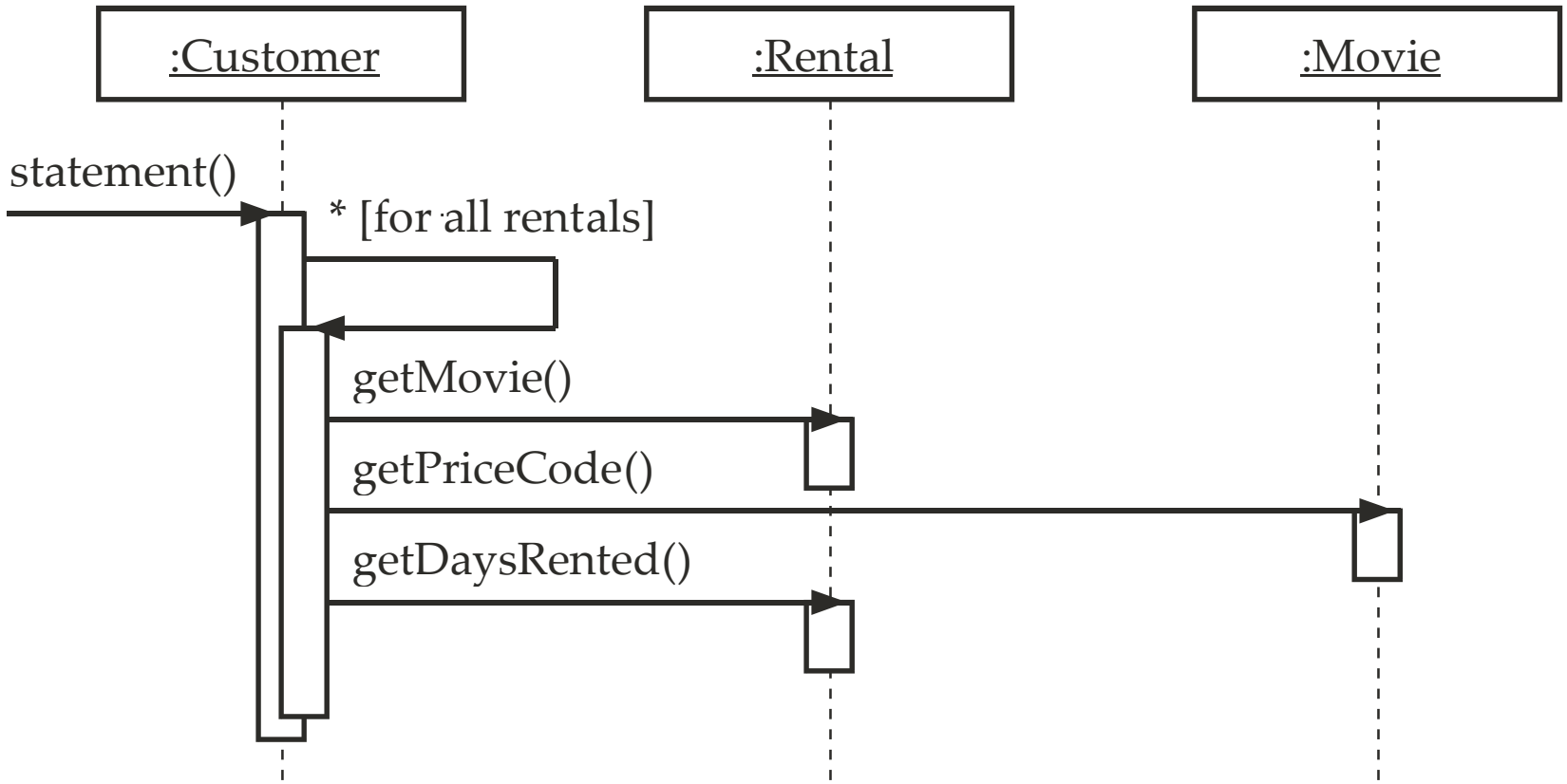
```
public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
      Enumeration rentals = _rentals.elements();
      String result = "Rental Record for " + getName() +     "\n";
      …
```

```
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;
    }
}
```

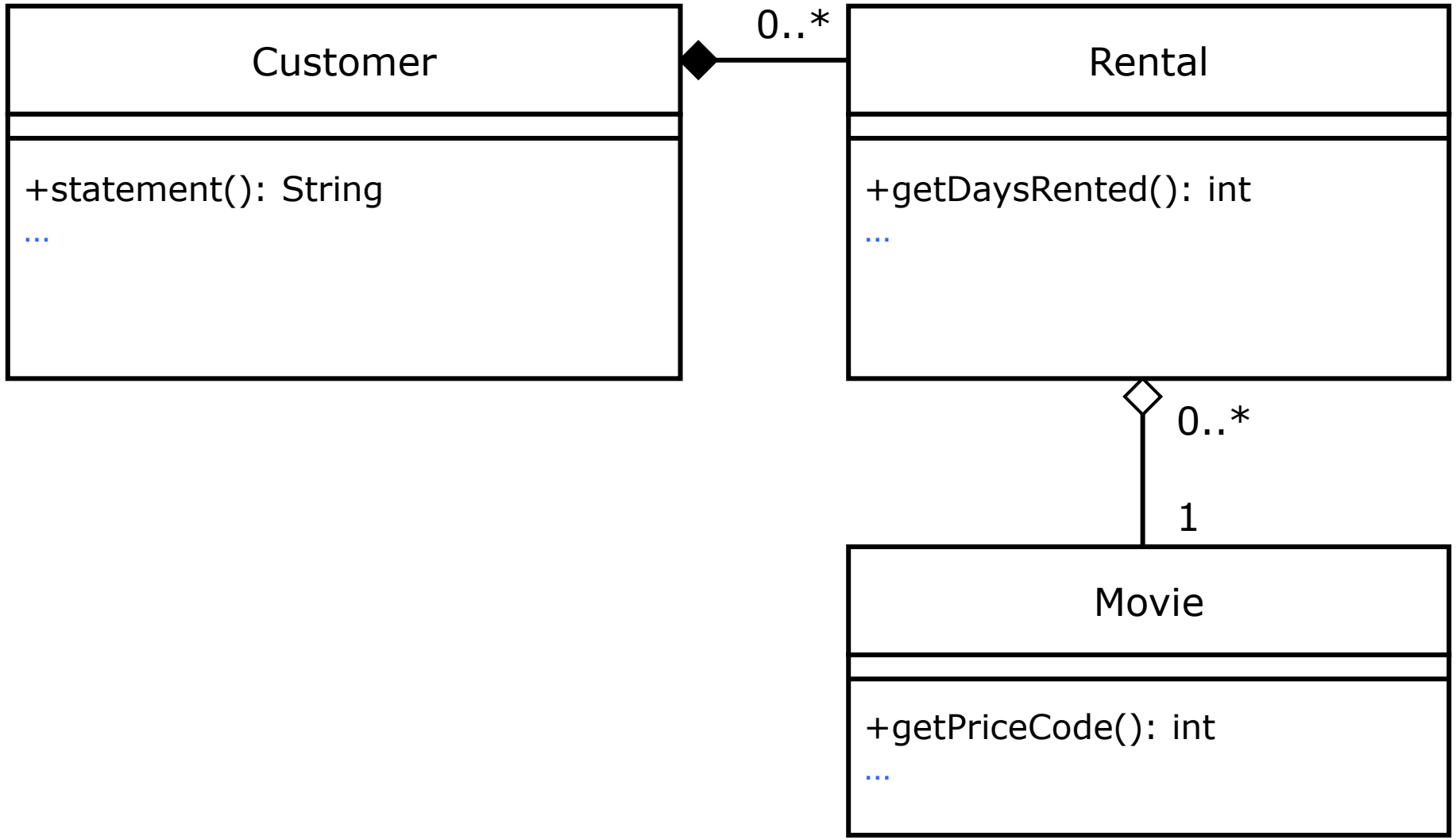
Replace Temp w/ Query (After)

```
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;
    }
}
```

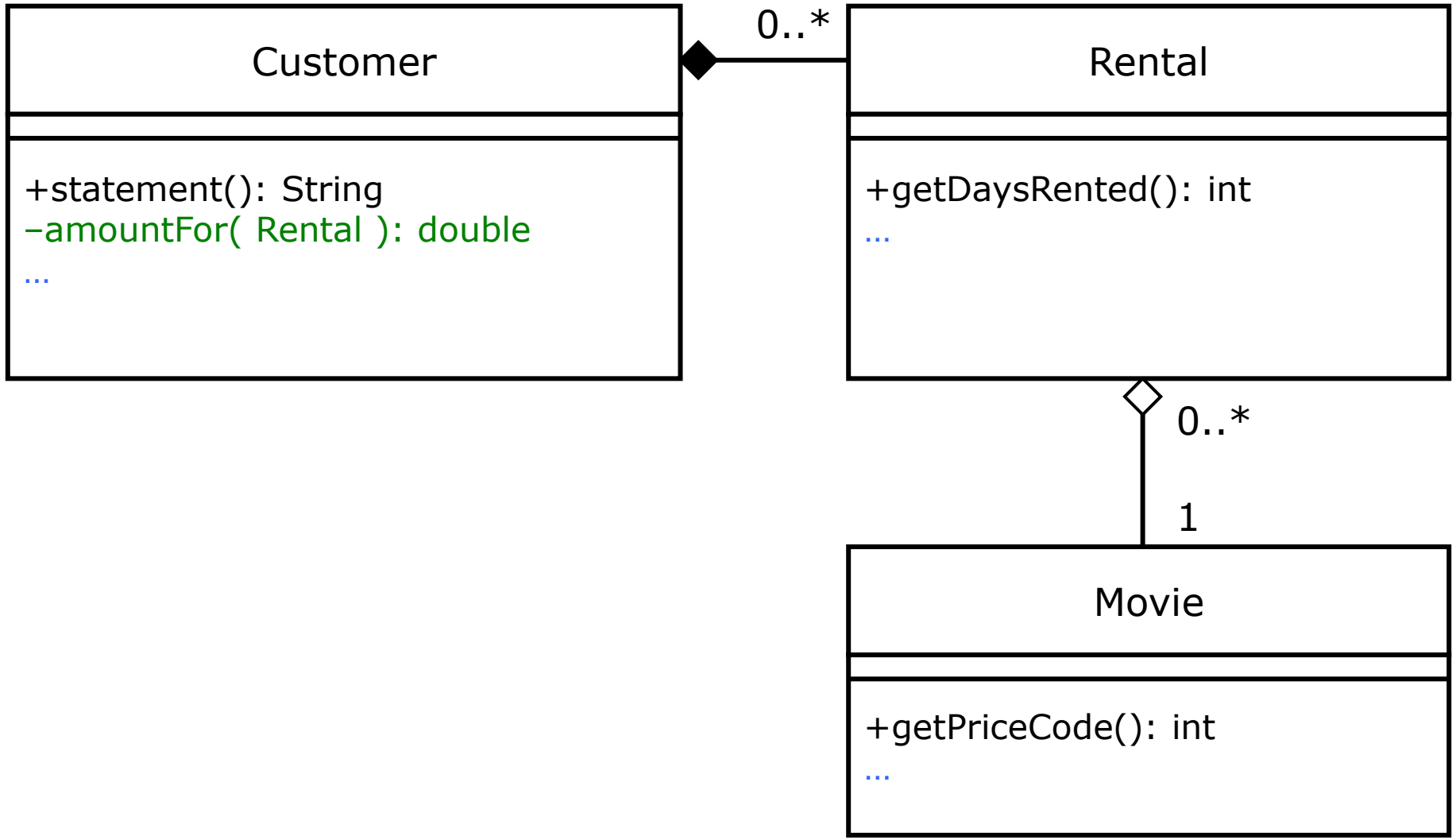
Replace Temp w/ Query (After)

```
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;
    }
    …
}
```

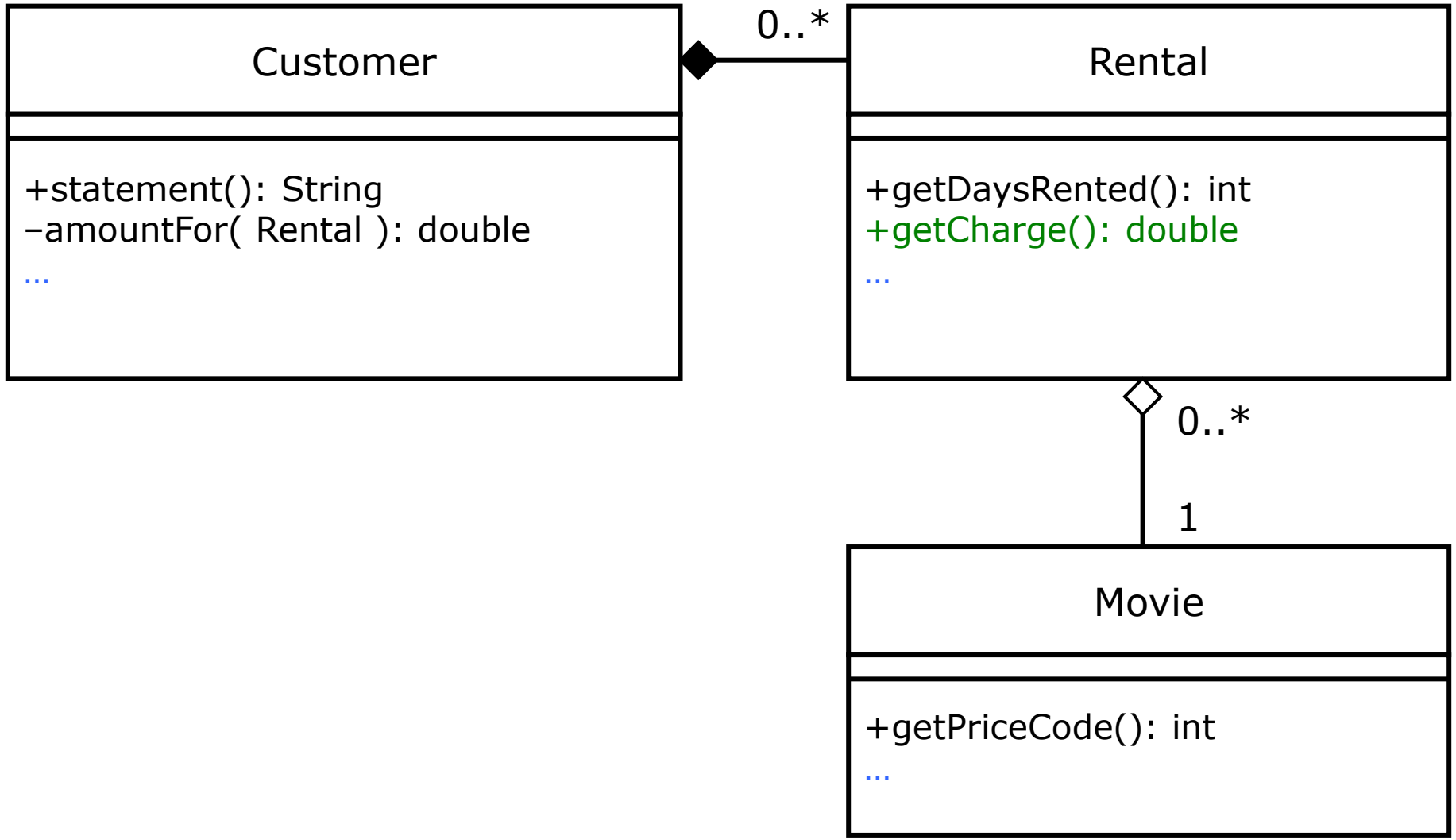
Initial Structural Design



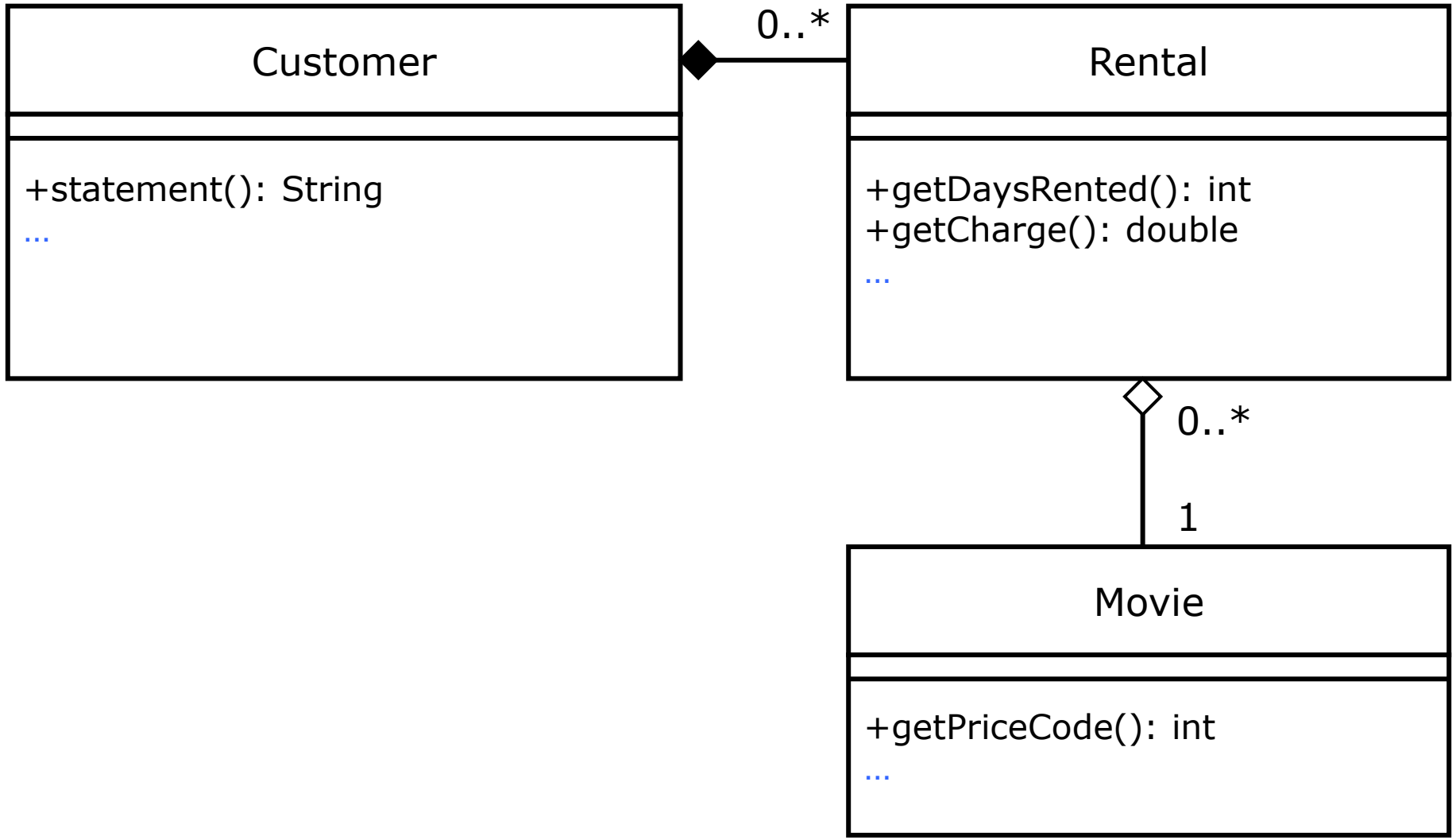
After Extract Method



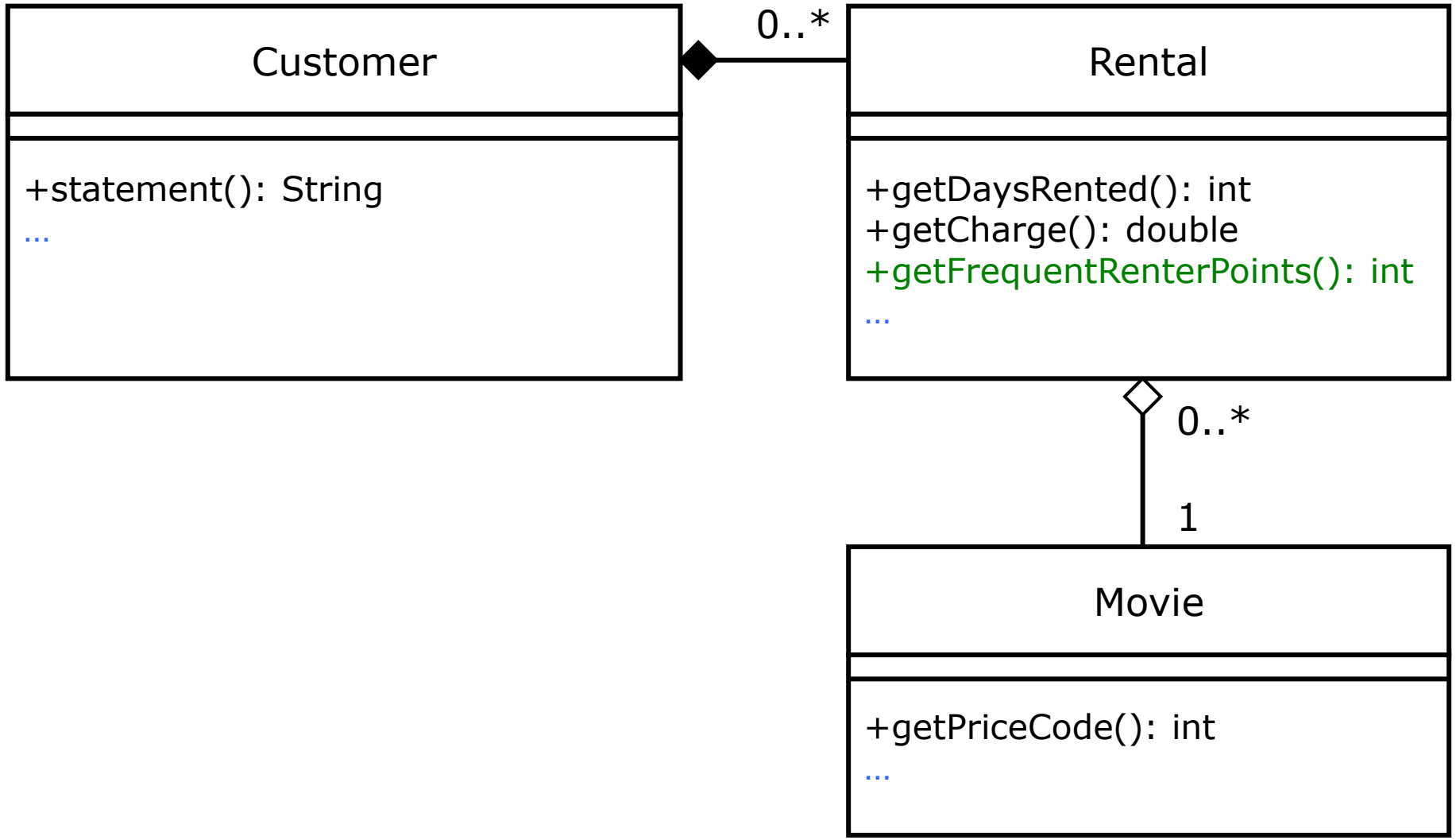
During Move Method



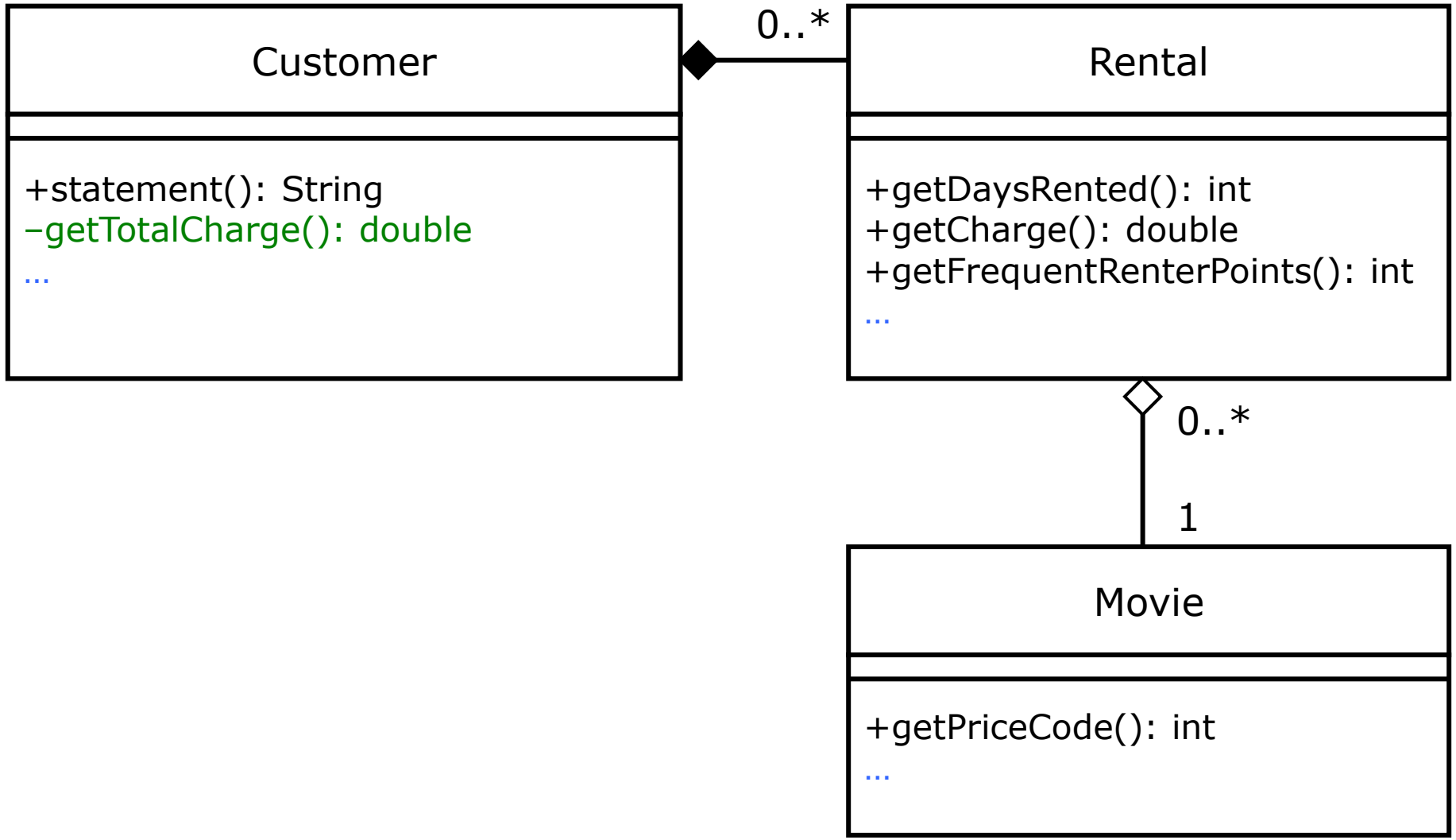
After Move Method



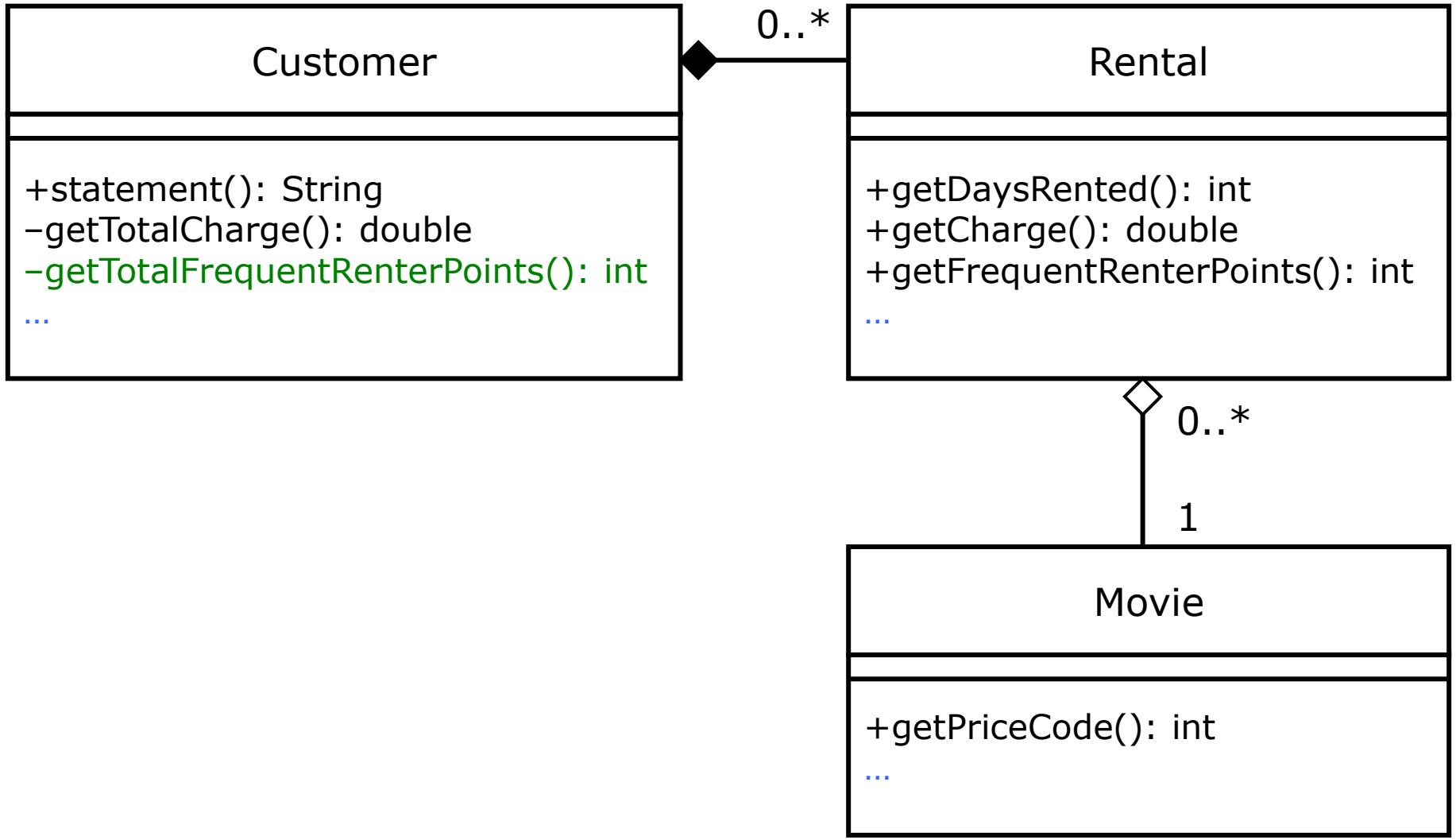
After Extract/Move Method



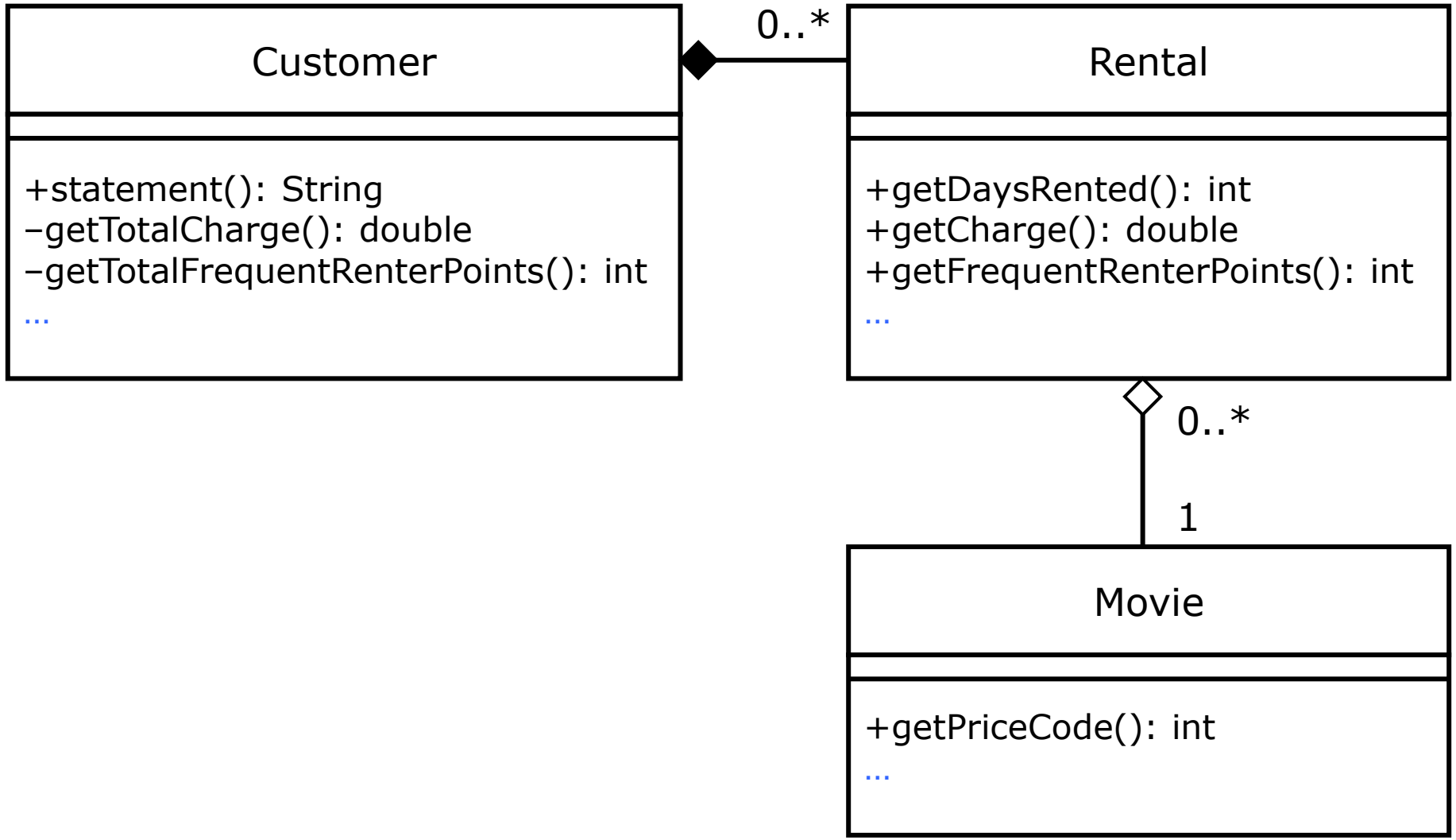
After Replace Temp w/ Query



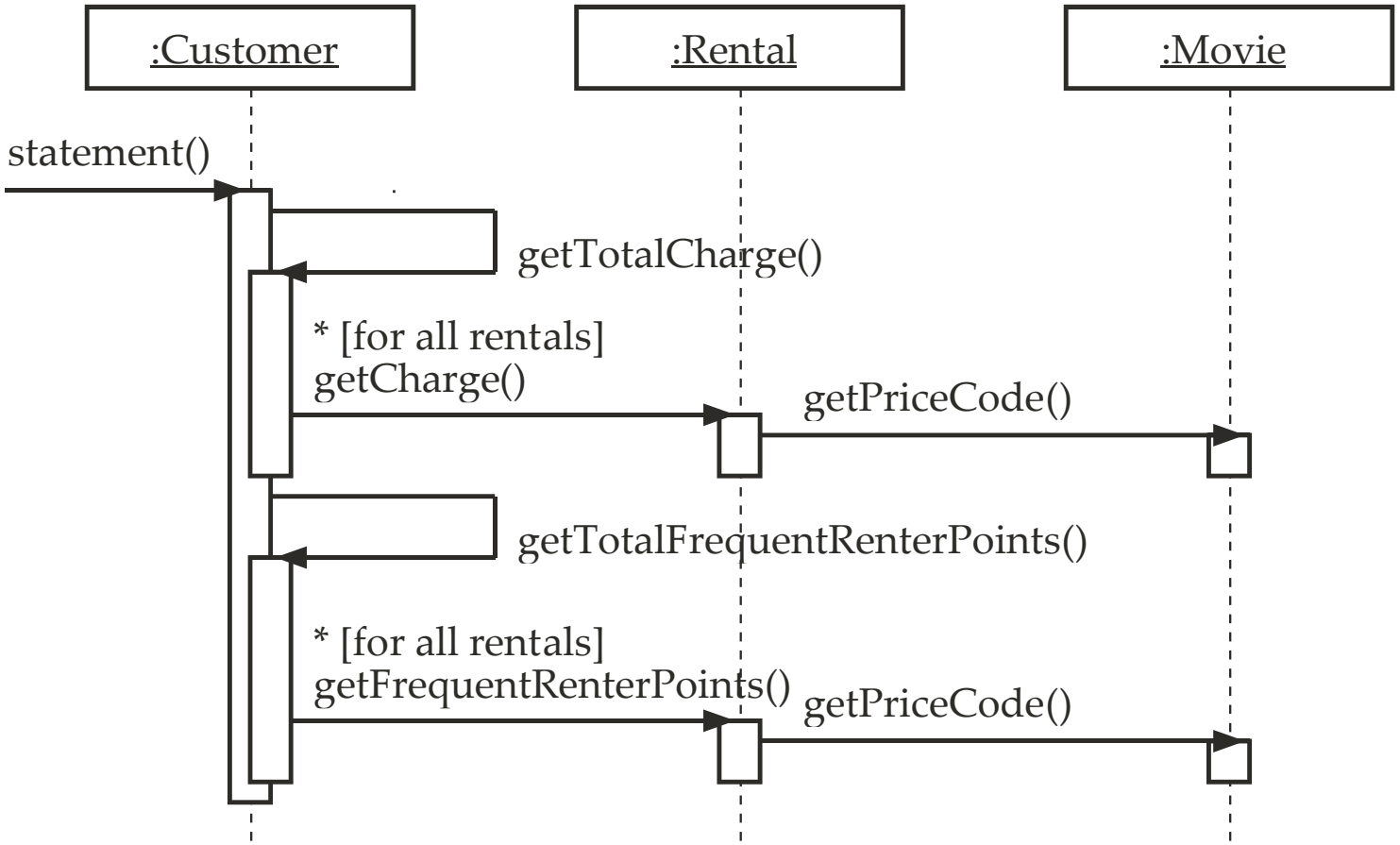
After Replace Temp w/ Query



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;
    }
    …
}
```

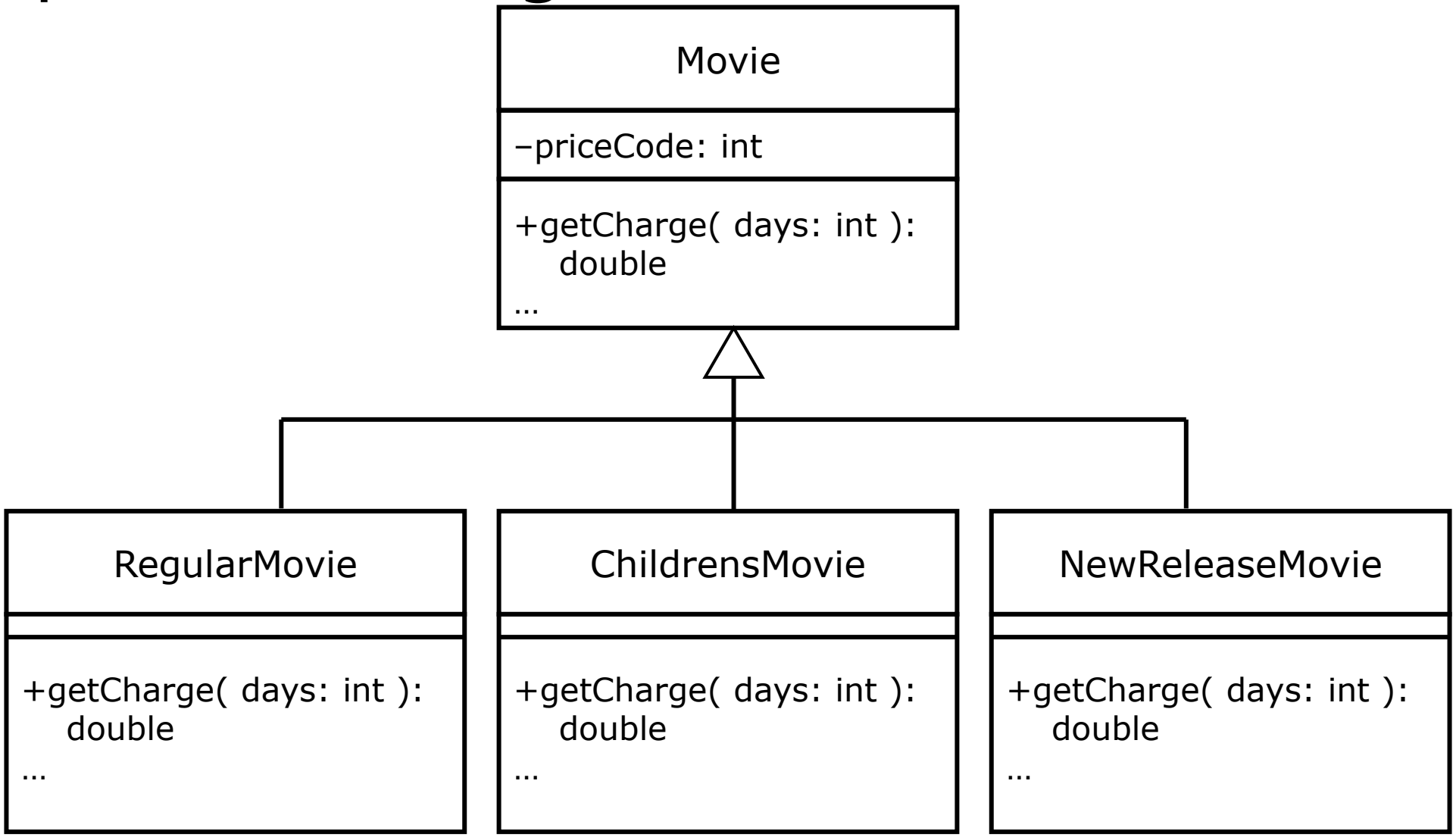
Move Method (After)

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

Replace Conditional Logic

- Ready for inheritance? …

Proposed Redesign?



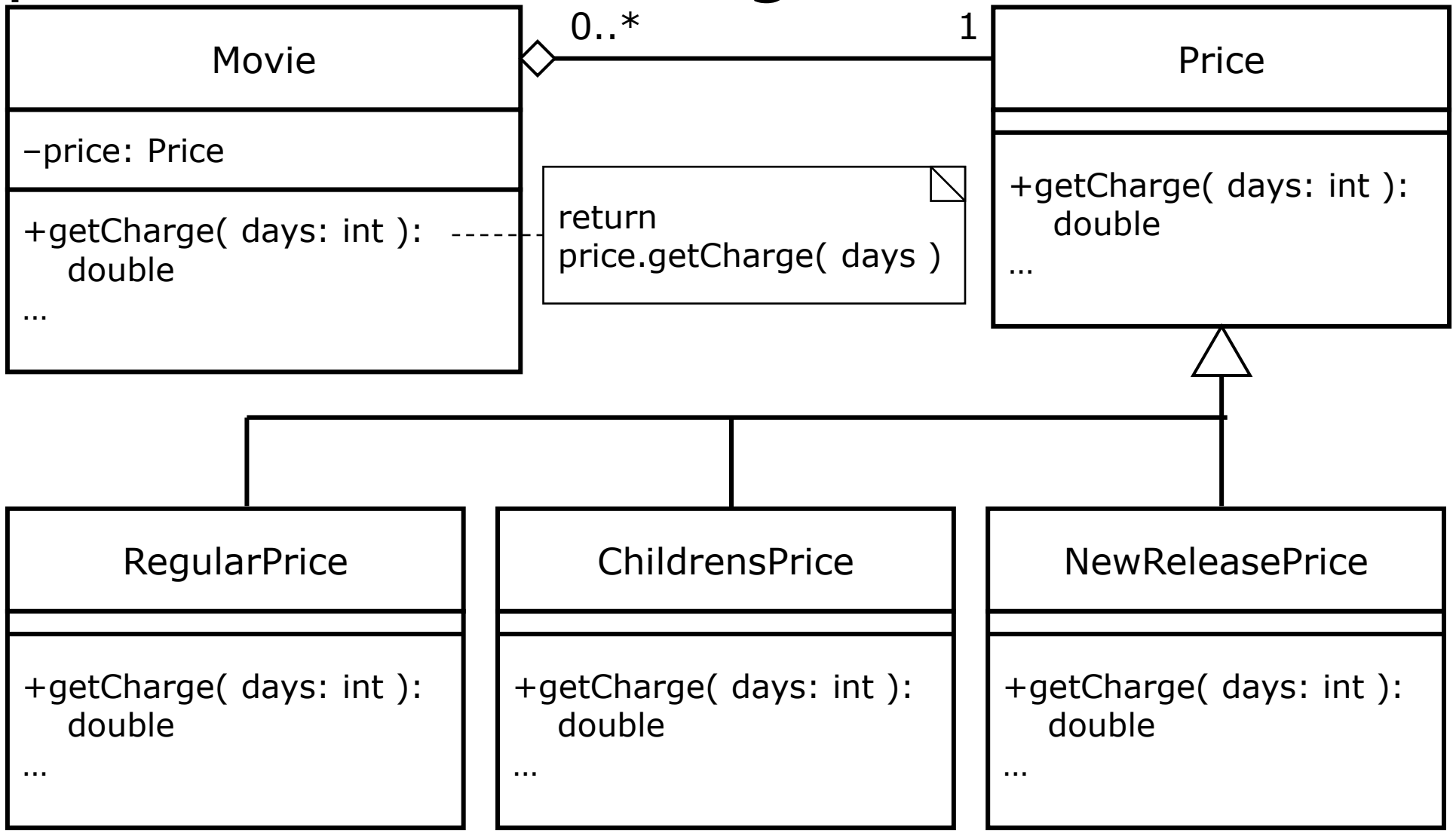
Proposed Redesign

- 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)

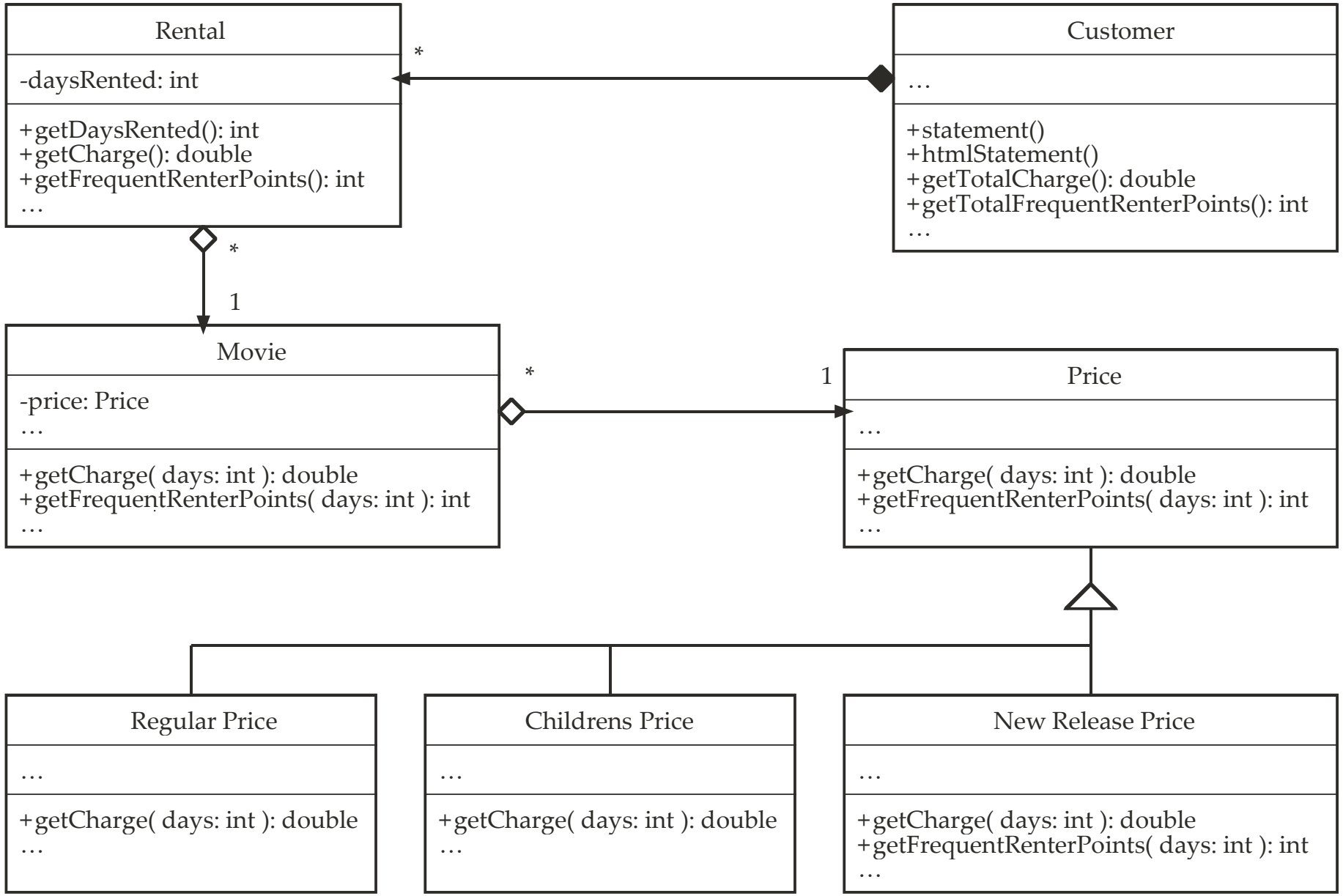
```
class Price {
    …
    public int getFrequentRenterPoints( int daysRented ) {
        if ((getPriceCode() == Movie.NEW_RELEASE) &&
            daysRented > 1)
            return 2;
        else
            return 1;
    }
    …
}
```

Replace Conditional with Polymorphism (After)

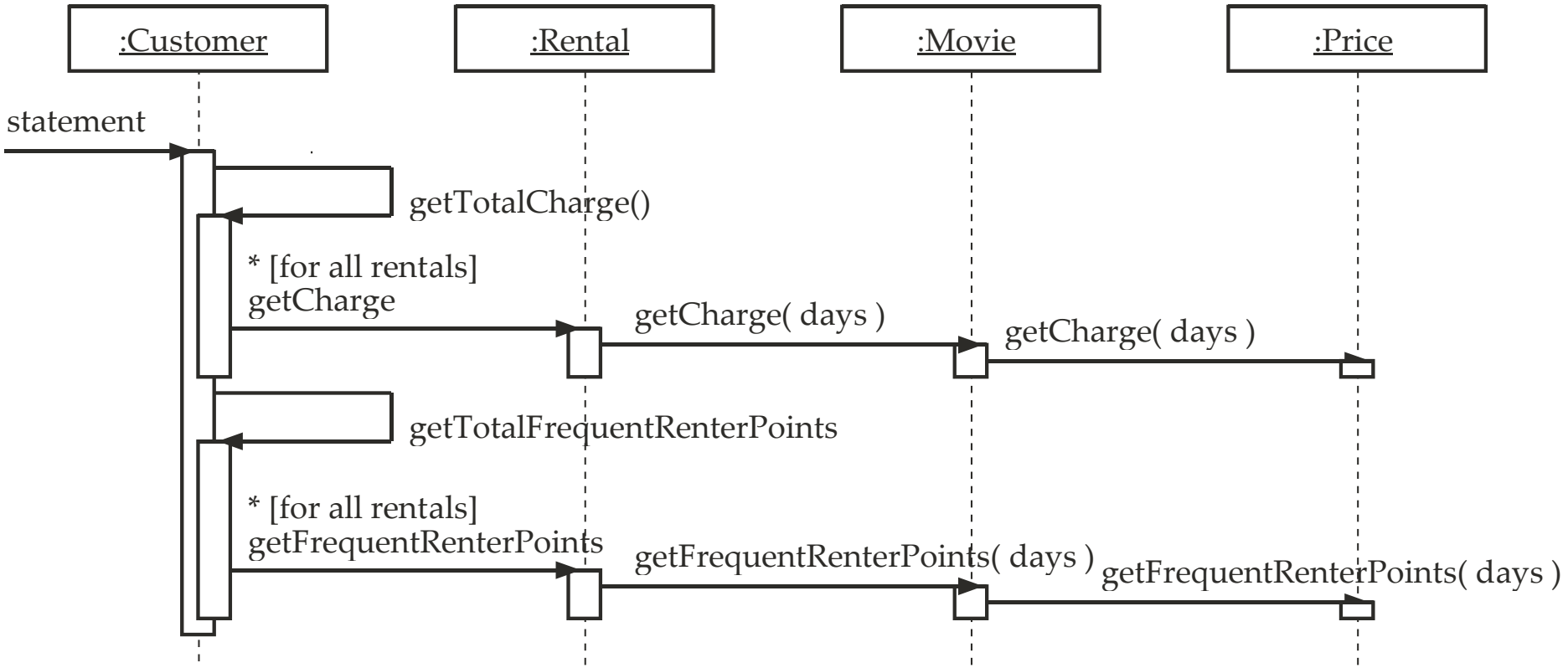
```
class Price {
    …
    public int getFrequentRenterPoints( int daysRented ) {
        return 1;
    }
    …
}

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


Rental
-daysRented: int
+getDaysRented(): int
+getCharge(): double
+getFrequentRenterPoints(): int
…
Customer
…
+statement()
+htmlStatement()
+getTotalCharge(): double
+getTotalFrequentRenterPoints(): int
…
*
*
1
Movie
-price: Price
…
+getCharge(days: int): double
+getFrequentRenterPoints(days: int): int
…
*
1
Price
…
+getCharge(days: int): double
+getFrequentRenterPoints(days: int): int
…
Regular Price
…
+getCharge(days: int): double
…
Childrens Price
…
+getCharge(days: int): double
…
New Release Price
…
+getCharge(days: int): double
+getFrequentRenterPoints(days: int): int
…

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 if you can meet the interest payments (extra maintenance costs)
  - If there is too much debt, you will be overwhelmed

    —Ward Cunningham

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

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
  - Practical Java
    - P. Haggar
    - Addison-Wesley, 2000

More Information

- Books:
  - 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
    - https://refactoring.com/