Refactoring

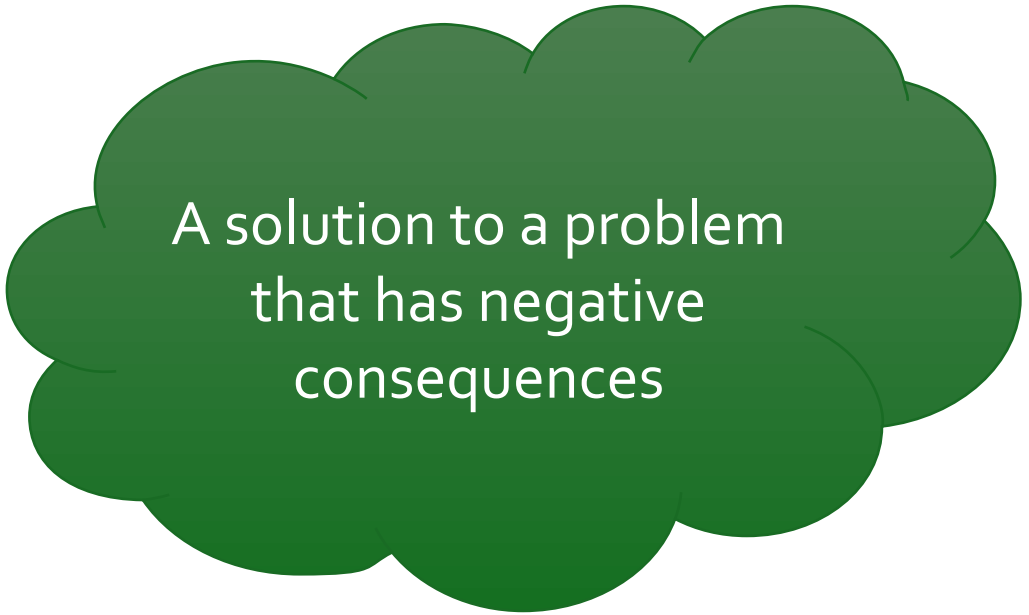
Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong, and Dr. Abram Hindle

Table of Contents

Introduction

Finding Solutions…



Pattern

Anti-pattern

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

Exercise

- What are some indicators or examples of poorly written code?

Bad Code 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

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

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

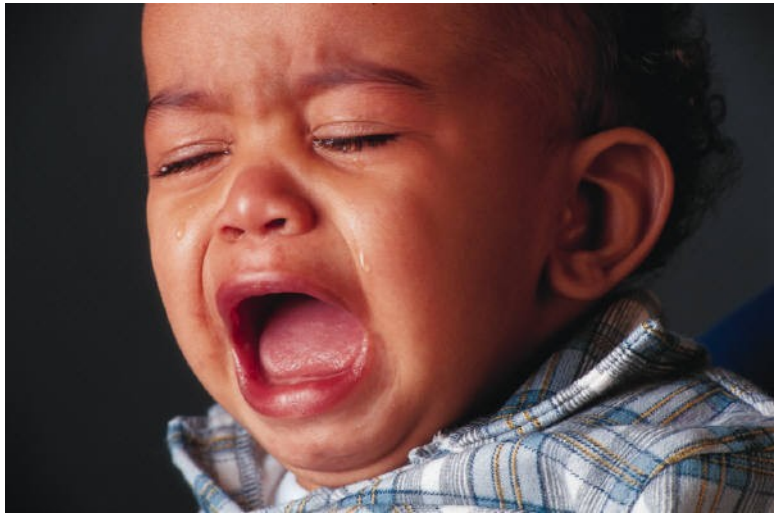
Kinds of Refactorings

- Dealing with generalization:
    - Getting methods and subclasses to the right place
    - Pull Up Method, Push Down Method, Extract Subclass, Extract Superclass

Code Smells

Bad Smells (in Code)

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

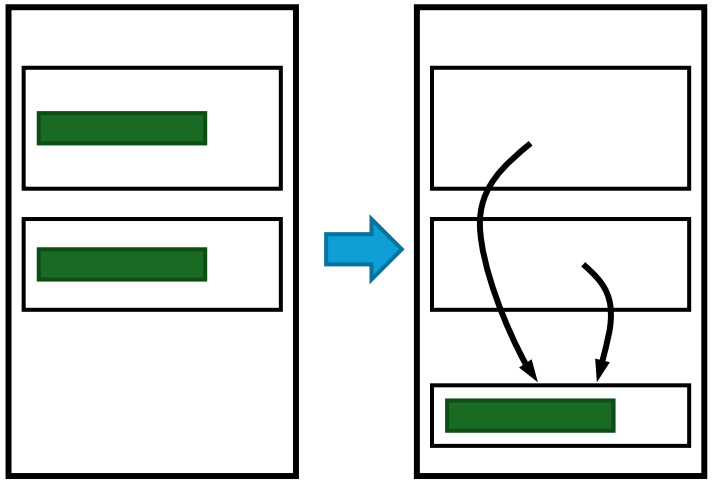


Bad Smells in Code

- 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

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

val 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

```
fun doSomething(x: Int, y: Int, z: Int){
    …
}
```

Primitive Obsession

- Indicator:
  - Using the built-in types too much
    - E.g., "stringitus", everything being a string

```
fun checkCode(postalCode: String) {
    …
}
```

- 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

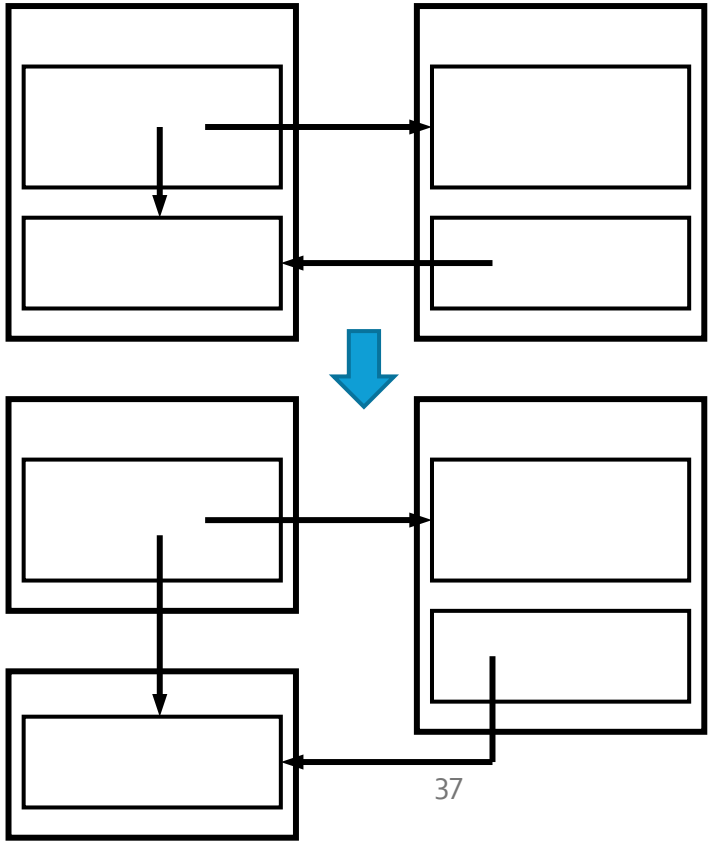
- Indicator:
  - Long chains of navigation to get to an object

```
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

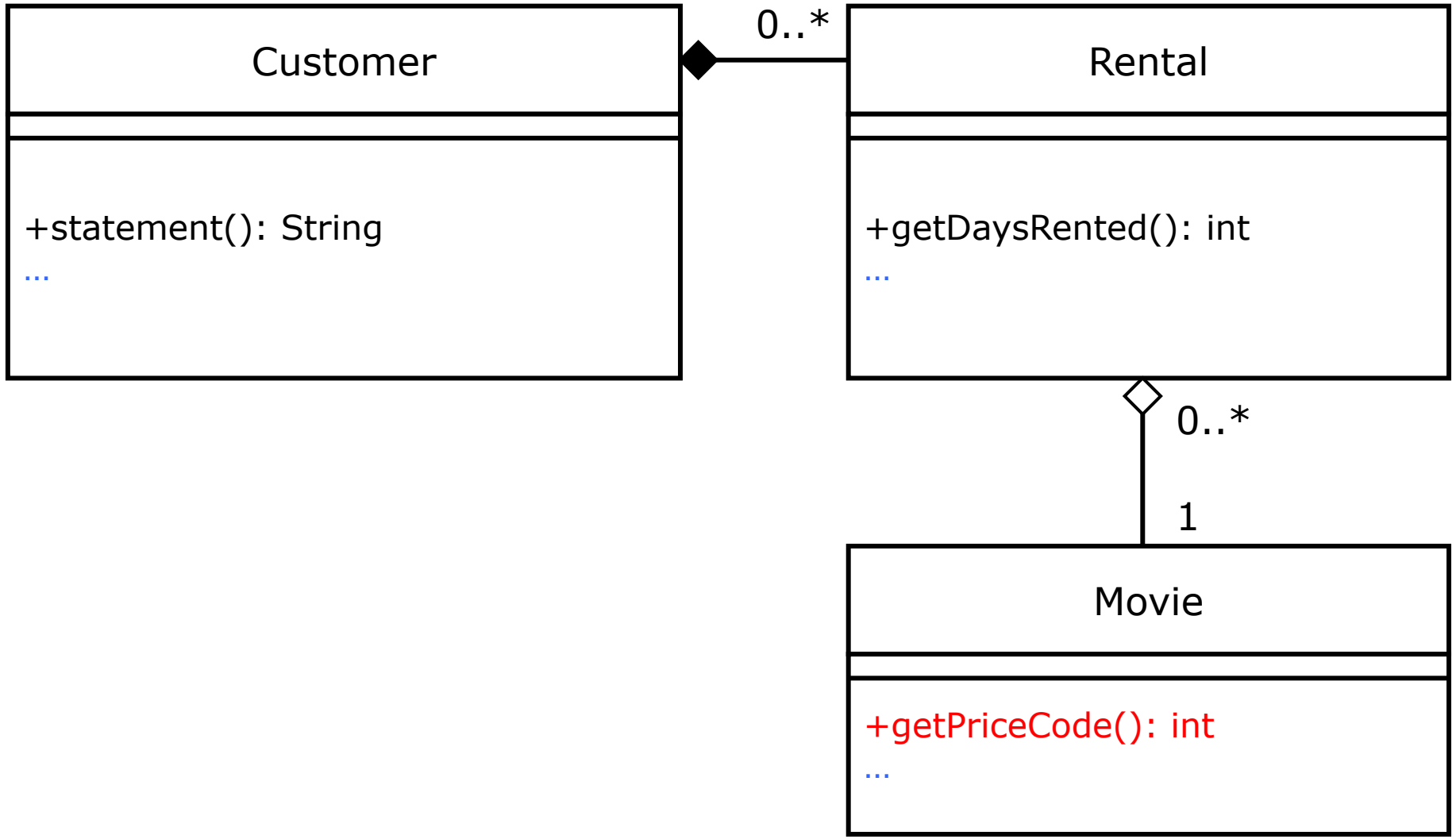
- Why:
  - Could be “deodorant” for bad smelling code
  - Simplify and refactor so comment is not needed
  - Use comments to explain *why* something was done a certain way
- Refactorings:
  - Extract Method
  - Rename Method

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 several days
    - Customer can collect frequent renter points
  - What kind of design?

Initial Structural Design



Initial Movie Class

```
class Movie(
      val title: String,
      var priceCode: Int
) {
      companion object {
         const val REGULAR = 0
         const val NEW_RELEASE = 1
         const val CHILDRENS = 2
      }
}
```

← companion object provides members that can be accessed through the class, similar to Java's static members.

Initial Rental Class

```
class Rental(
      val movie: Movie,
      val daysRented: Int
)
```

Initial Customer Class

```
class Customer(val name: String) {
   private val rentals = mutableListOf<Rental>()

   fun addRental(rental: Rental) {
      rentals.add(rental);
   }
   …
```

```
fun statement(): String {
   var totalAmount = 0.0;
   var frequentRenterPoints = 0;
   var result = StringBuilder()
   result.append("Rental Record for $name\n")
   …
```

Still in Customer class

```
// determine amounts for each line
for (rental in rentals) {
      var thisAmount = 0.0
      when (rental.movie.priceCode) {
            Movie.REGULAR -> {
                  thisAmount += 2.0
            }
            Movie.NEW_RELEASE -> {
                  thisAmount += rental.daysRented * 3.0
            }
            Movie.CHILDRENS -> {
                  thisAmount += 1.5
            }
      }
}
…
```

Still in statement method of Customer class

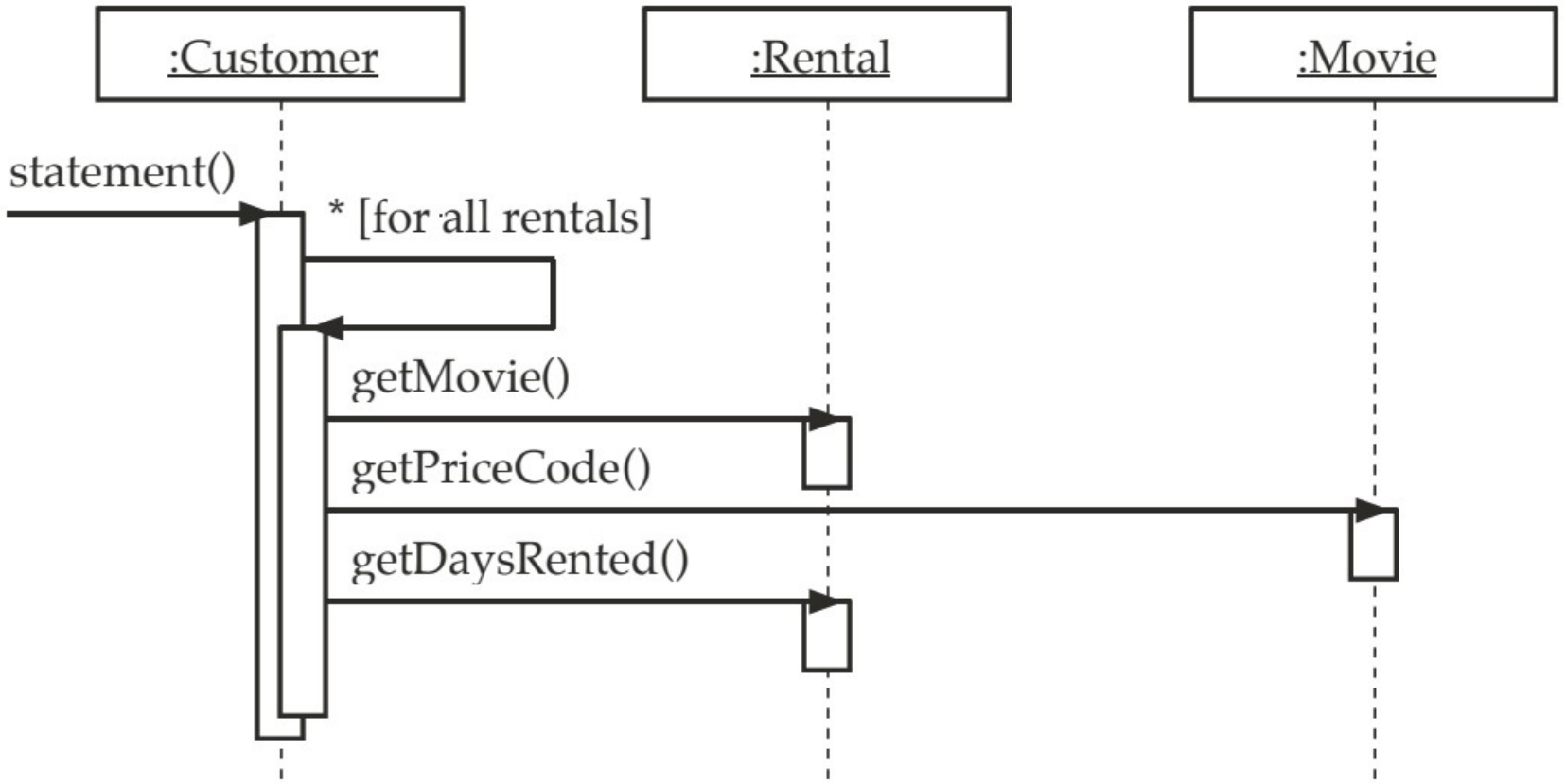
```
    // add frequent renter points
    frequentRenterPoints++;

    // add bonus for new release rental
    if (rental.movie.priceCode == Movie.NEW_RELEASE &&
    rental.daysRented > 1) {
       frequentRenterPoints++
    }

    // show figures for this rental
    result.append("\t${rental.movie.title}\t$thisAmount\n")
    totalAmount += thisAmount;
}
...
```

Still in statement method of Customer class

```
    // add footer lines
       result.append("Amount owed is $totalAmount\n")
       result.append("You earned $frequentRenterPoints
              frequent renter points")

       return result.toString()
    }
}
```

Still in statement method of Customer class

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
    - We will make changes in small steps
    - Do test-driven development
        - Before each change we should make and run unit tests
        - Make the small change
        - Then test again

Refactoring

- Goal:
    - Decompose `statement()` method
    - We will do these by extracting logical chunks of code as a new methods

Refactoring Roadmap

Too much code in `statement()`

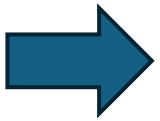
1. First, we will *extract* the code that determines movie costs as a new method
2. Next, we will *move* the extracted method to `Rental` class
   - 2.5: Plus, we will update temporary variable usage
3. Then, we will *extract* the code that calculates frequent renter reward points as a new method in the `Rental` class
4. Finally, we will *extract* temporary variable usage as new methods

1. Extract Movie Costs Code

- First step before extracting the code:
    - Make and run unit tests

1. Extract Movie Costs Code

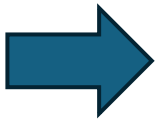
```
when (rental.movie.priceCode) {
    Movie.REGULAR -> {
            thisAmount += 2.0
     }
        Movie.NEW_RELEASE -> {
            thisAmount += rental.daysRented * 3.0
        }
        Movie.CHILDRENS -> {
            thisAmount += 1.5
        }
  }
  …
```



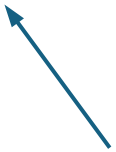
```
thisAmount = amountFor(rental)
```

1. Extract Movie Costs Code

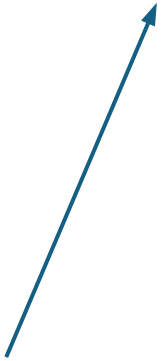
```
when (rental.movie.priceCode) {
    Movie.REGULAR -> {
            thisAmount += 2.0
     }
         Movie.NEW_RELEASE -> {
             thisAmount += rental.daysRented * 3.0
         }
         Movie.CHILDRENS -> {
             thisAmount += 1.5
         }
  }
  …
```



```
thisAmount = amountFor(rental)
```



We will take this code



And put it in this function we will define

1. Extract Movie Costs Code

```
class Customer(val name: String) {
     …
     private fun amountFor(rental: Rental): Double {
          var thisAmount = 0.0
          when (rental.movie.priceCode) {
               Movie.REGULAR -> {
                         thisAmount += 2.0
                }
                    Movie.NEW_RELEASE -> {
                         thisAmount += rental.daysRented * 3.0
                    }
                    Movie.CHILDRENS -> {
                         thisAmount += 1.5
                    }
           }
           return thisAmount
      }
```

The code to determine the movie cost is now taken out of the large method and defined as a new method

1. Extract Movie Costs Code

- After extracting, we compile and test
    - Remember, make small steps and test often!

2. Move amountFor() Method

- Refactoring:
  - Move `amountFor()` to `Rental` class
    - Method uses *rental* information, but not *customer* information
  - Move this method to the right class

2. Move amountFor() Method (Before)

```
class Customer(val name: String) {
    …
    private fun amountFor(rental: Rental): Double {
        var thisAmount = 0.0
        when (rental.movie.priceCode) {
            Movie.REGULAR -> {
                thisAmount += 2.0
            }
            Movie.NEW_RELEASE -> {
                thisAmount += rental.daysRented * 3.0
            }
            Movie.CHILDRENS -> {
                thisAmount += 1.5
            }
        }
        return thisAmount
    }
```

2. Move amountFor() Method (Before)

```
class Customer(val name: String) {
     …
     private fun amountFor(rental: Rental): Double {
          var thisAmount = 0.0
          when (rental.movie.priceCode) {
               Movie.REGULAR -> {
                         thisAmount += 2.0
                }
                    Movie.NEW_RELEASE -> {
                         thisAmount += rental.daysRented * 3.0
                    }
                    Movie.CHILDRENS -> {
                         thisAmount += 1.5
                    }
           }
           return thisAmount
      }
```

Feature envy! Customer class just using data from rental class

Code in orange will be *naming changes* for better readability when we move the method to the Rental class

2. Move amountFor() Method (After)

```
class Rental(val movie: Movie, val daysRented: Int){
    …
    fun getCharge(): Double {
        var result = 0.0
        when (movie.priceCode) {
            Movie.REGULAR -> {
                result += 2.0
            }
            Movie.NEW_RELEASE -> {
                result += daysRented * 3.0
            }
            Movie.CHILDRENS -> {
                result += 1.5
            }
        }
        return result
    }
```

2. Move amountFor() Method (After)

```
class Customer(val name: String){
    …
    private fun amountFor(rental: Rental) {
        return rental.getCharge()
    }
    …
}
```

For now, we update amountFor() in the Customer class.
However, we will update usage in statement() to use rental.getCharge() instead, and eventually remove amountFor().

2. Move amountFor() Method (Cleaning Up)

- Again, compile and test
- Then, we will clean up the indirection
  - Customer class still using `amountFor()`, which just calls `rental.getCharge()`
  - We can just call `rental.getCharge()` directly instead
  - Then, we can remove `amountFor()`

2. Move amountFor() Method (Cleaning Up)

```
fun statement() {
    …
    for (rental in rentals) {
      thisAmount = amountFor(rental)
      …
  }
```

2. Move amountFor() Method (Cleaning Up)

```
fun statement() {
    …
    for (rental in rentals) {
      thisAmount = rental.getCharge()
      …
  }
```

2. Move amountFor() Method (Cleaning Up)

- Once again, compile and test

2. Move amountFor() Method (Cleaning Up)

- Refactoring:
    - Eliminate `thisAmount` temporary in `statement()`
    - Replace redundant temporary variable with query

2.5 Update Temp Variable Usage (Before)

```
class Customer(val name: String){
    …
    fun statement(): String {
        …

        for (rental in rentals){
            var thisAmount = 0
            thisAmount = rental.getCharge()

            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((rental.movie.priceCode == Movie.NEW_RELEASE) &&
                rental.daysRented() > 1){ frequentRenterPoints++ }

            // show figures for this rental
            result.append("\t${rental.movie.title}\t$thisAmount\n")
            totalAmount += thisAmount
        }
        …
```

2.5 Update Temp Variable Usage (After)

```
class Customer(val name: String){
    …
    class statement(): String {
        …

   for (rental in rentals){
            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((rental.movie.priceCode == Movie.NEW_RELEASE) &&
                rental.daysRented() > 1){ frequentRenterPoints++ }

            // show figures for this rental
            val amount = rental.getCharge()
            result.append("\t${rental.movie.title}\t$amount\n")
            totalAmount += amount
        }
        …
```

3. Extract Renter Points Code

- Refactoring:
  - Similar to what we did with the movie costs code, extract frequent renter reward points logic from `statement()`
    - Applicable rules go with the *`Rental`* class, not *`Customer`*
  - We will continue to write and run unit tests as we make these changes (though it will not be explicitly stated)

3. Extract Renter Points Code (Before)

```
class Customer(val name: String){
    …
    class statement(): String {
        …


    for (rental in rentals){
            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((rental.movie.priceCode == Movie.NEW_RELEASE) &&
                rental.daysRented() > 1){ frequentRenterPoints++ }

        // show figures for this rental
        val amount = rental.getCharge()
            result.append("\t${rental.movie.title}\t$amount\n")
            totalAmount += amount
        }
        …
```

3. Extract Renter Points Code (After)

```
class Customer(val name: String){
    …
    class statement(): String {
        …

    for (rental in rentals){
            // add frequent renter points
            frequentRenterPoints += rental.getFrequentRenterPoints()

            // show figures for this rental
        val amount = rental.getCharge()
            result.append("\t${rental.movie.title}\t$amount\n")
            totalAmount += amount
        }
        …
```

← We will move the code into this function we will define

3. Extract Renter Points Code (After)

```
class Rental(val movie: Movie, val daysRented: Int) {
    …
    fun getFrequentRenterPoints(): Int {
        return if (movie.priceCode == Movie.NEW_RELEASE && daysRented > 1) 2 else 1
    }
}
```

- We define the method in the Rental class directly, since we already updated the code in the Customer class to use this method
- The logic in the method is a condensed if else statement

4. Replace Temp with Query 1

- Refactoring:
  - Eliminate the temporary variable `totalAmount` and replace with `getTotalCharge()` query

4. Replace Temp with Query 1 (Before)

```
class Customer(val name: String){
    …
    class statement(): String {
     var totalAmount = 0.0;
     var frequentRenterPoints = 0;
     var result = StringBuilder()

     result.append("Rental Record for $name\n")


      for (rental in rentals){
             // add frequent renter points
             frequentRenterPoints += rental.getFrequentRenterPoints()

             // show figures for this rental
          val amount = rental.getCharge()
             result.append("\t${rental.movie.title}\t$amount\n")
             totalAmount += amount
          }


     // add footer lines

     result.append("Amount owed is $totalAmount\n")

          result.append("You earned $frequentRenterPoints frequent renter points")

          return result.toString()

     }
}
```

4. Replace Temp with Query 1 (After)

```
class Customer(val name: String){
    …
    class statement(): String {
     var frequentRenterPoints = 0;
     var result = StringBuilder()

     result.append(“Rental Record for $name\n”)


      for (rental in rentals){
             // add frequent renter points
             frequentRenterPoints += rental.getFrequentRenterPoints()

             // show figures for this rental
          val amount = rental.getCharge()
             result.append(“\t${rental.movie.title}\t$amount\n”)

          }


     // add footer lines

     result.append(“Amount owed is ${getTotalCharge()}\n”)

          result.append(“You earned $frequentRenterPoints frequent renter points”)

          return result.toString()

     }
}
```

4. Replace Temp with Query 1 (After)

```
class Customer(val name: String){
    …
    private fun getTotalCharge(): Double {
        var result = 0.0;
    for (rental in rentals) {
            result += rental.getCharge()
        }
        return result
    }
    …
}
```

We make a private function that calculates total charge that statement() can use

4. Replace Temp with Query 2

- Refactoring:
    - Eliminate `frequentRenterPoints` temporary and replace with `getTotalFrequentRenterPoints()` query

4. Replace Temp with Query 2 (Before)

```
class Customer(val name: String){
    …
    class statement(): String {
     var frequentRenterPoints = 0;
     var result = StringBuilder()

     result.append(“Rental Record for $name\n”)


      for (rental in rentals){
             // add frequent renter points
             frequentRenterPoints += rental.getFrequentRenterPoints()

             // show figures for this rental
          val amount = rental.getCharge()
             result.append(“\t${rental.movie.title}\t$amount\n”)

          }


     // add footer lines

     result.append(“Amount owed is ${getTotalCharge()}\n”)

          result.append(“You earned $frequentRenterPoints frequent renter points”)

          return result.toString()

     }
}
```

4. Replace Temp with Query 2 (After)

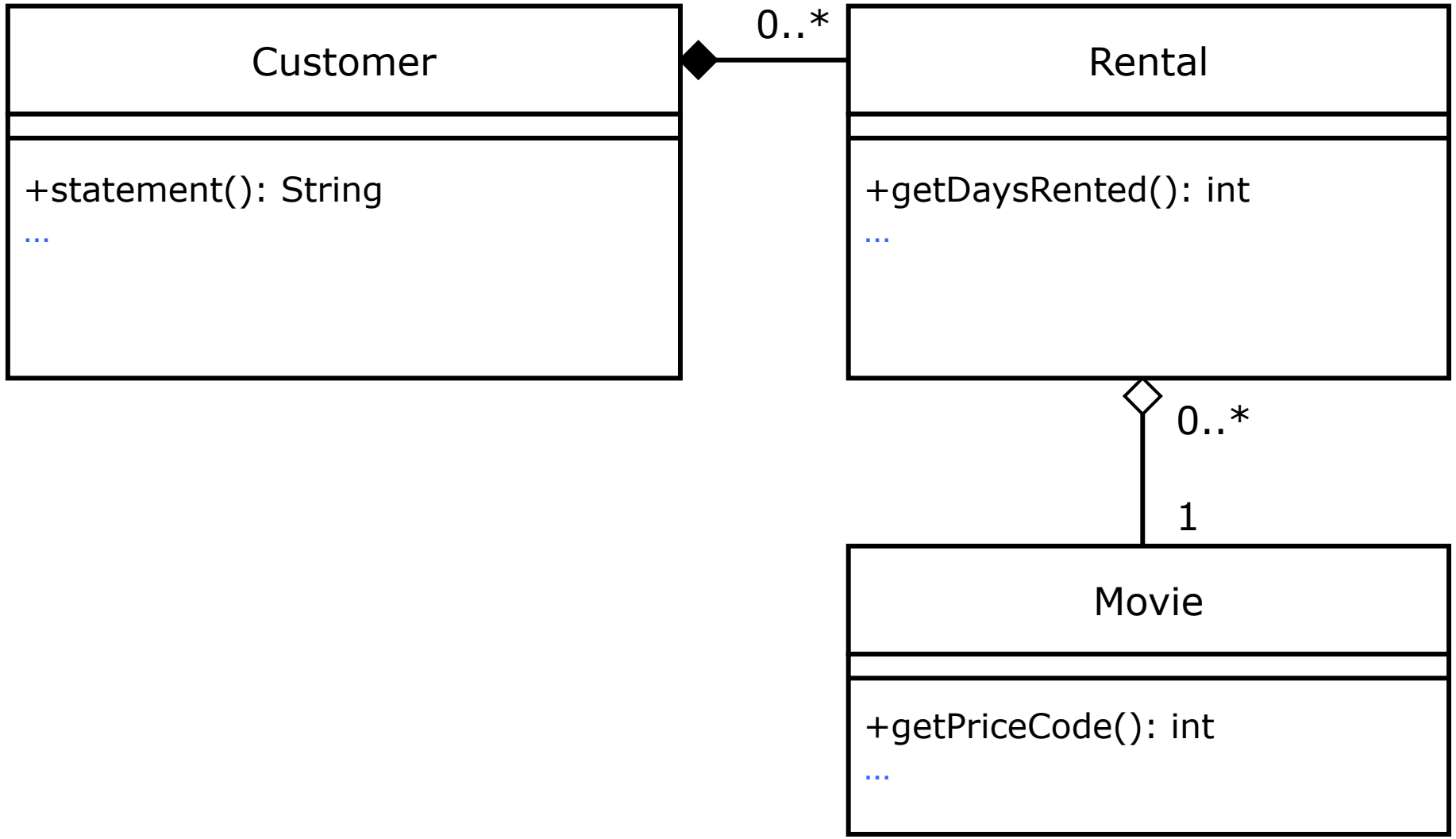
```
class Customer(val name: String){
    …
    class statement(): String {
    var result = StringBuilder()

    result.append("Rental Record for $name\n")


     for (rental in rentals){
            // show figures for this rental
         val amount = rental.getCharge()
            result.append("\t${rental.movie.title}\t$amount\n")

         }


    // add footer lines

    result.append("Amount owed is ${getTotalCharge()}\n")

        result.append("You earned ${getTotalFrequentRenterPoints()} frequent renter points")

        return result.toString()

    }
}
```

4. Replace Temp with Query 2 (After)

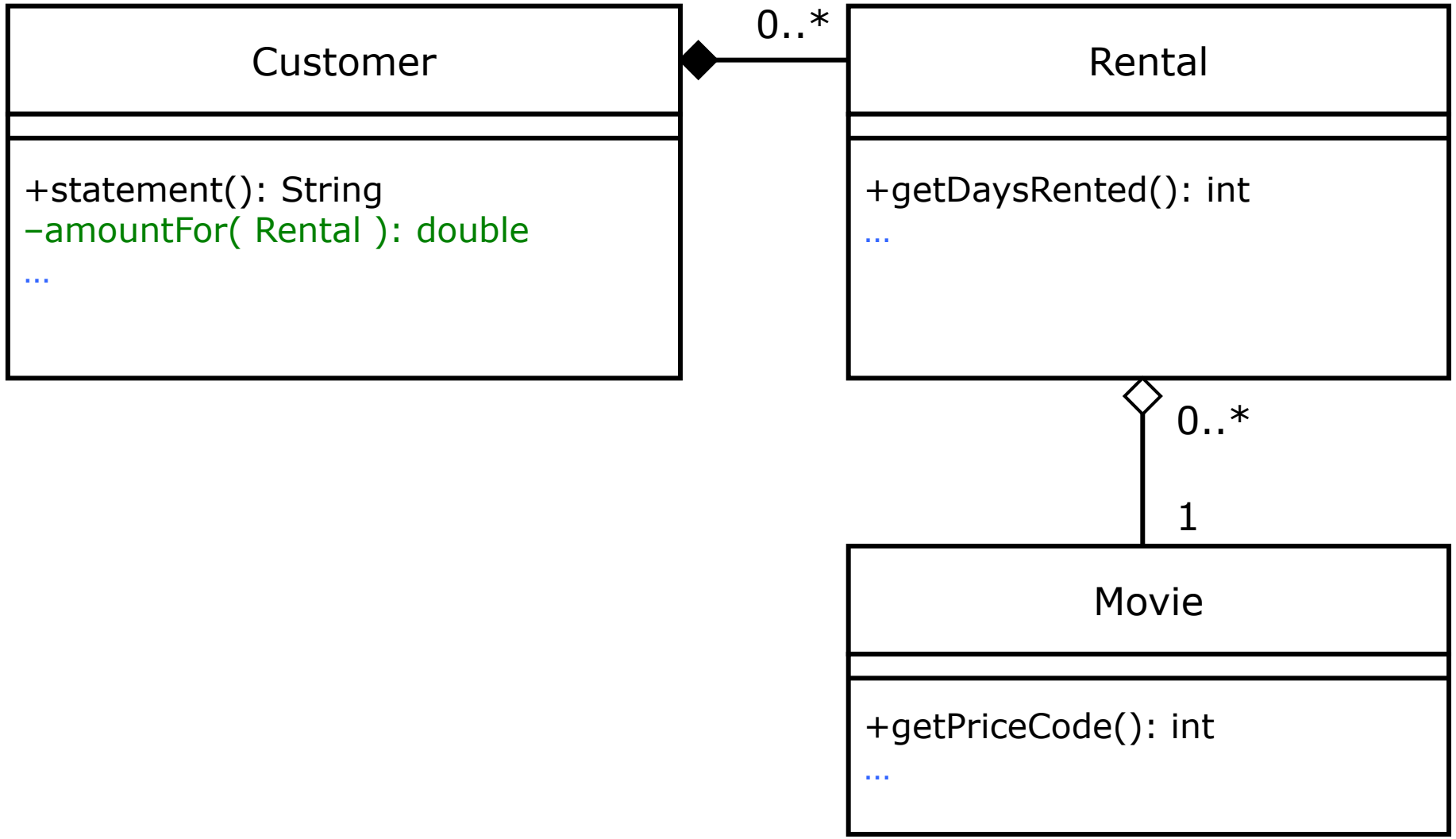
```
class Customer(val name: String){
    …
    private fun getTotalFrequentRenterPoints(): Int {
        var result = 0;
        for (rental in rentals) {
            result += rental.getFrequentRenterPoints()
        }
        return result;
    }
    …
}
```

Similarly, we make another private function that calculates total renter points that statement() can use

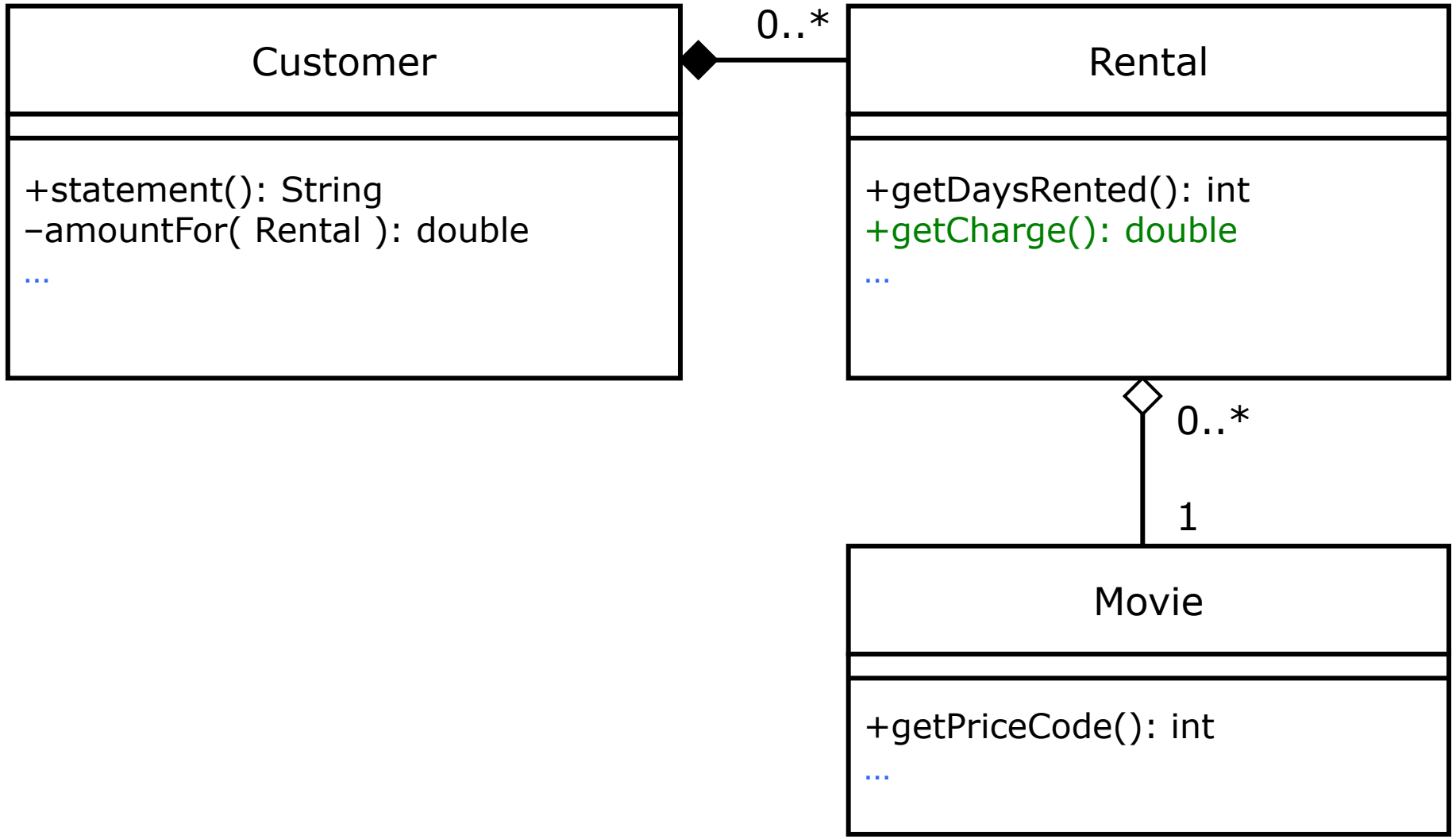
Initial Structural Design



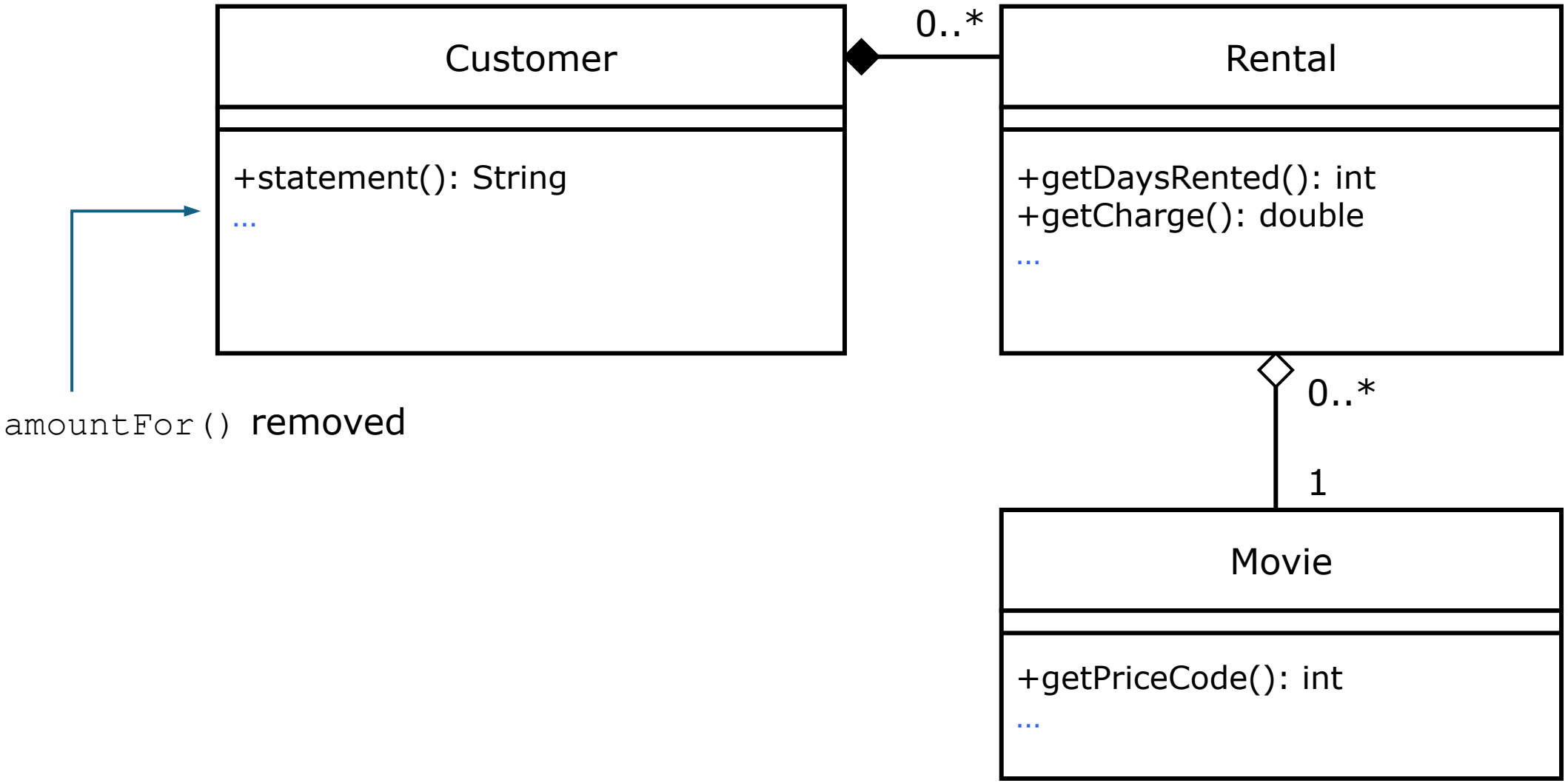
After Extracting Movie Costs Code



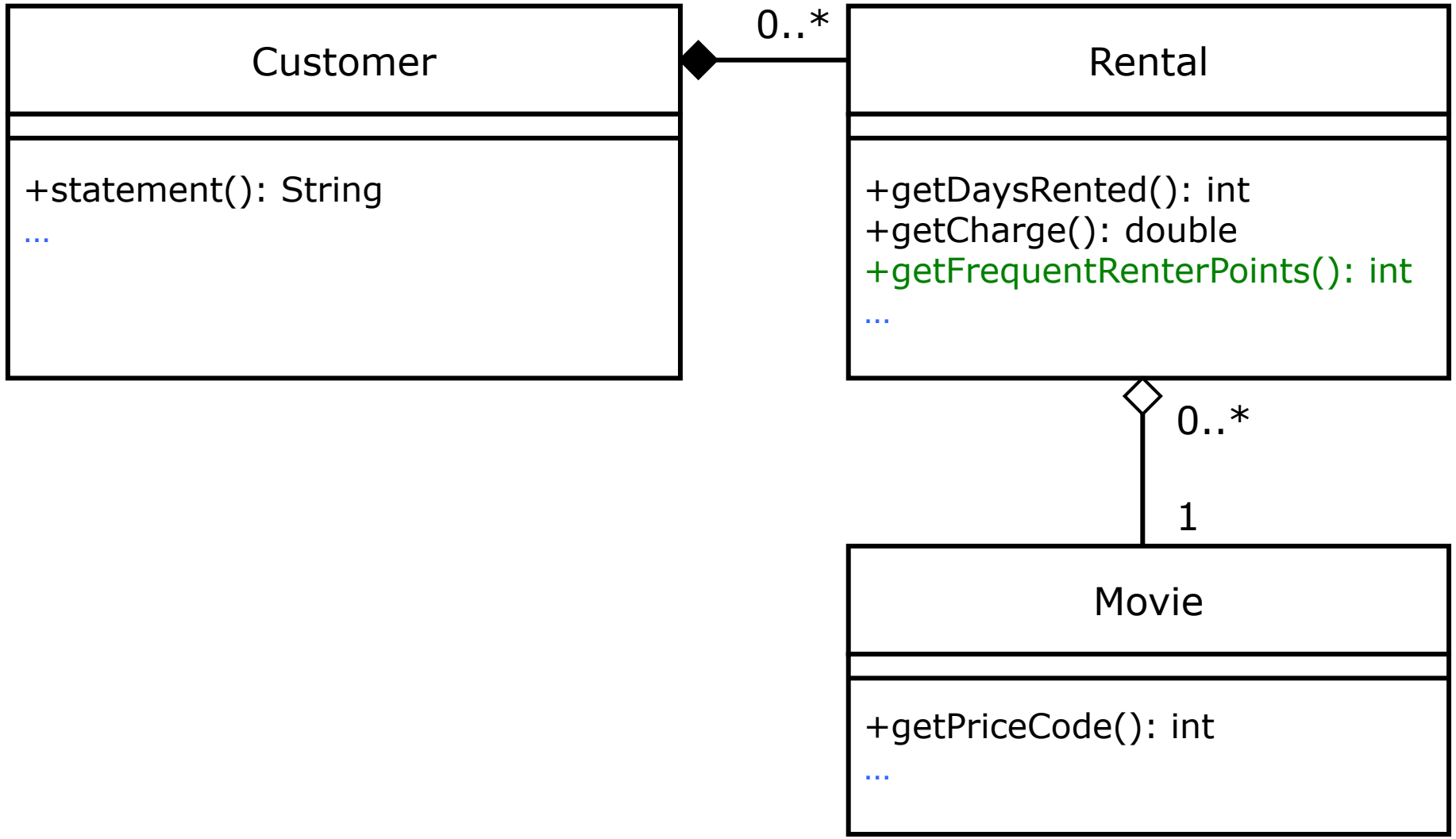
During Moving amountFor() Method



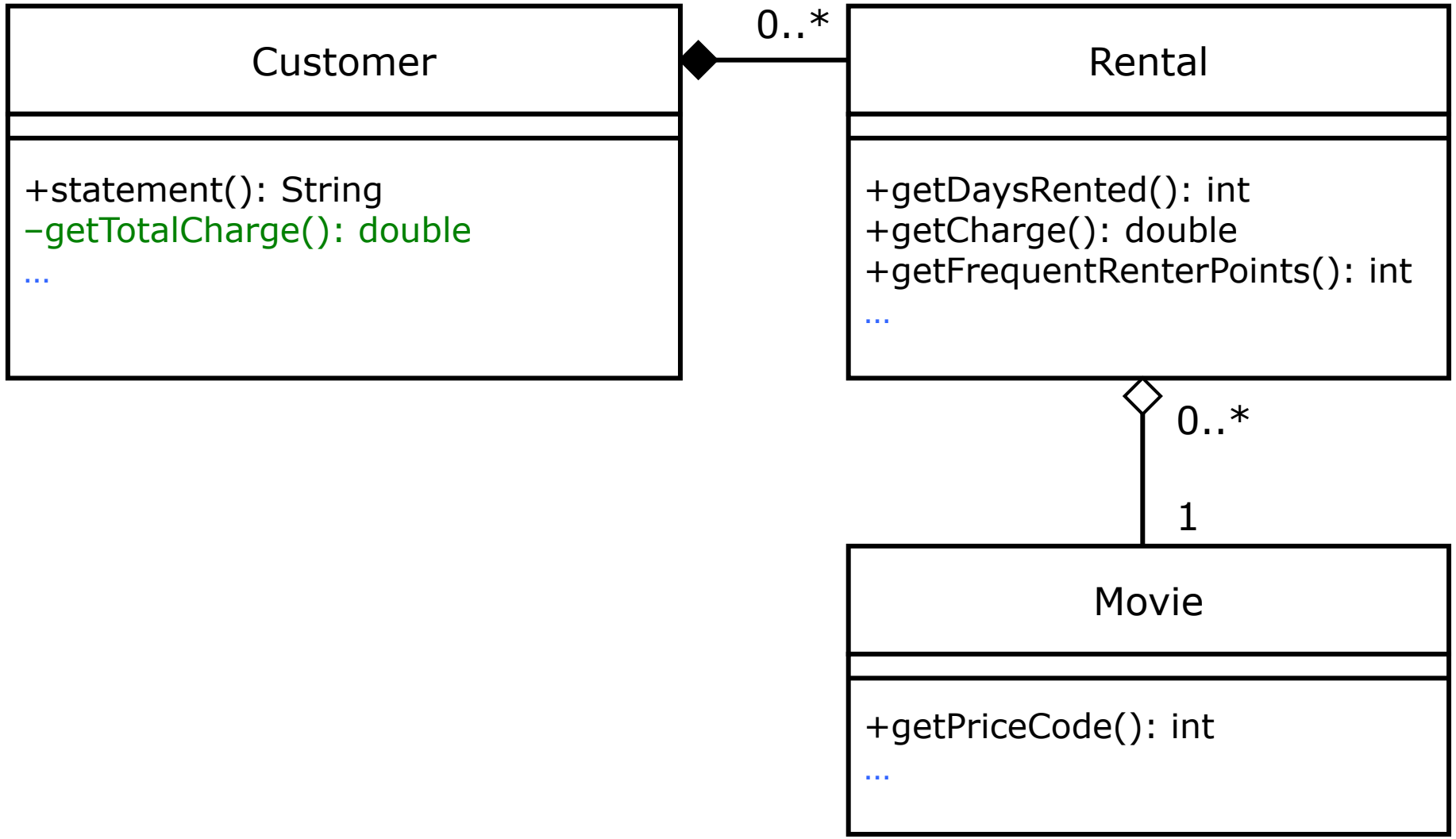
After Moving amountFor() Method



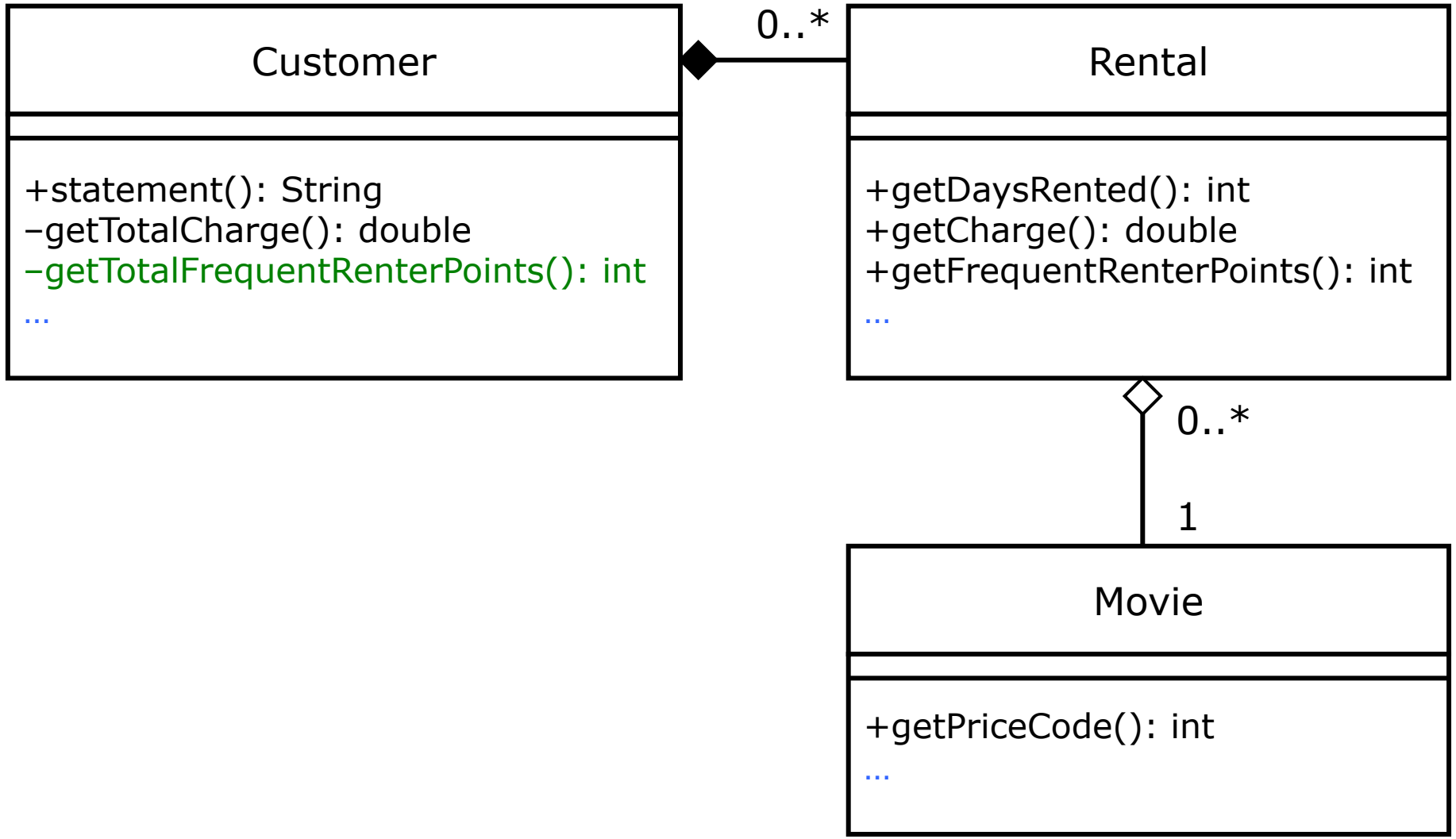
After Extracting and Moving Renter Points Code



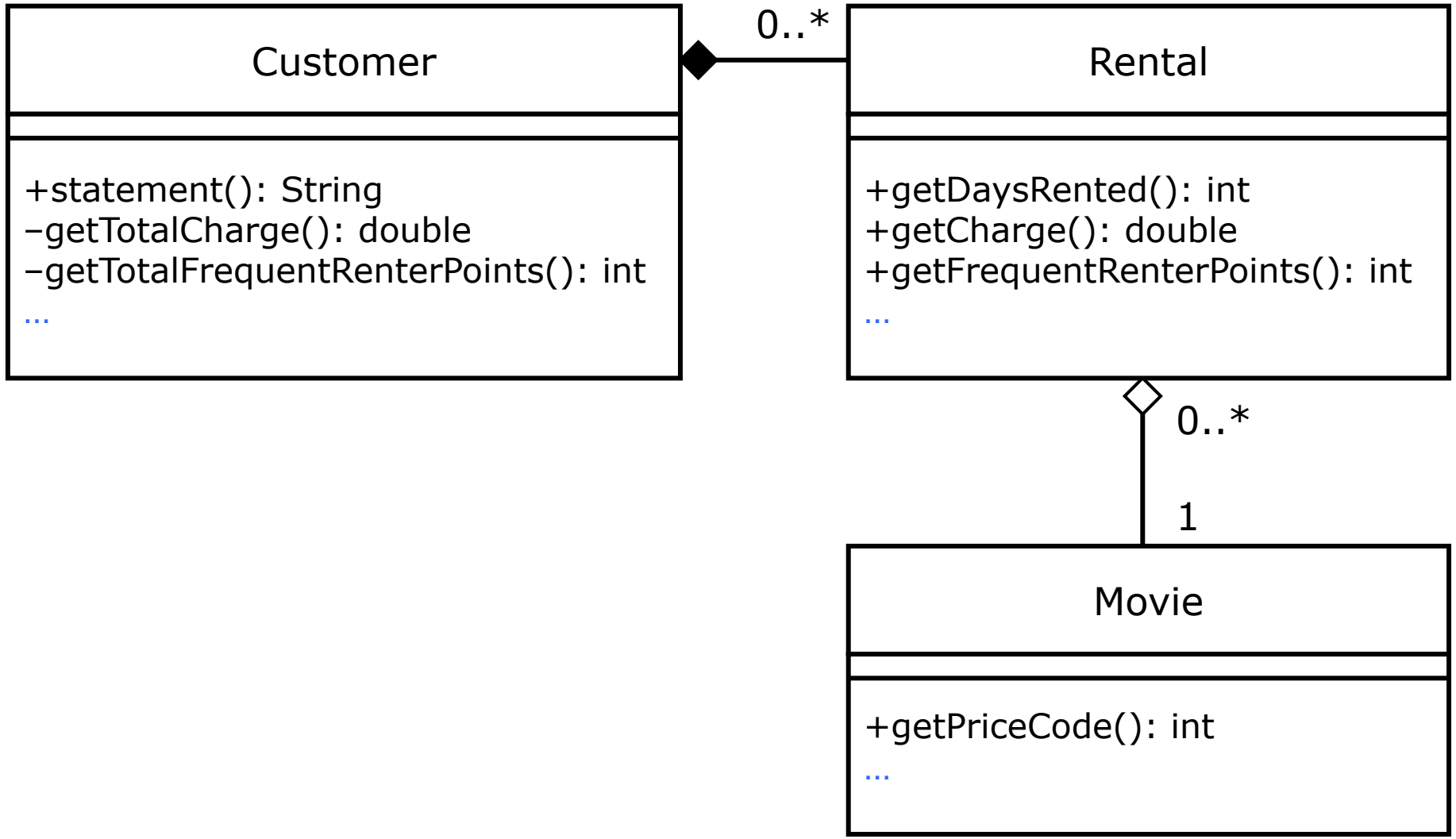
After Replacing Temp with Query 1



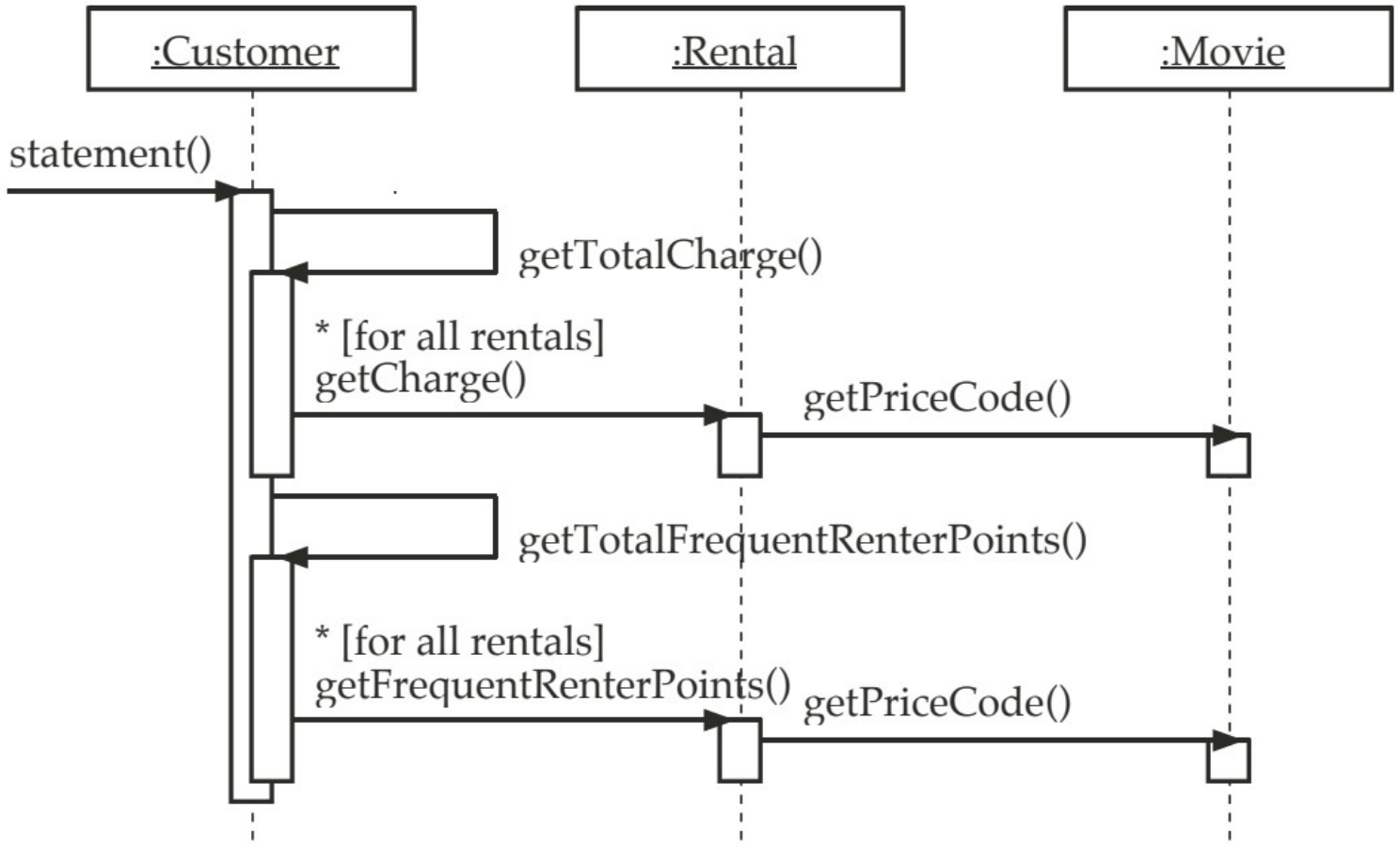
After Replacing Temp with Query 2



Initial Refactor Structural Design



Initial Refactor Behavioral Design



Refactoring

- Consequences:
    - More object-oriented
        - Decomposes big methods into smaller ones
        - Distributes responsibilities among classes
        - Easier to add new features
            - E.g., HTML output
            - E.g., new charging rules
    - More code
    - Slower performance?

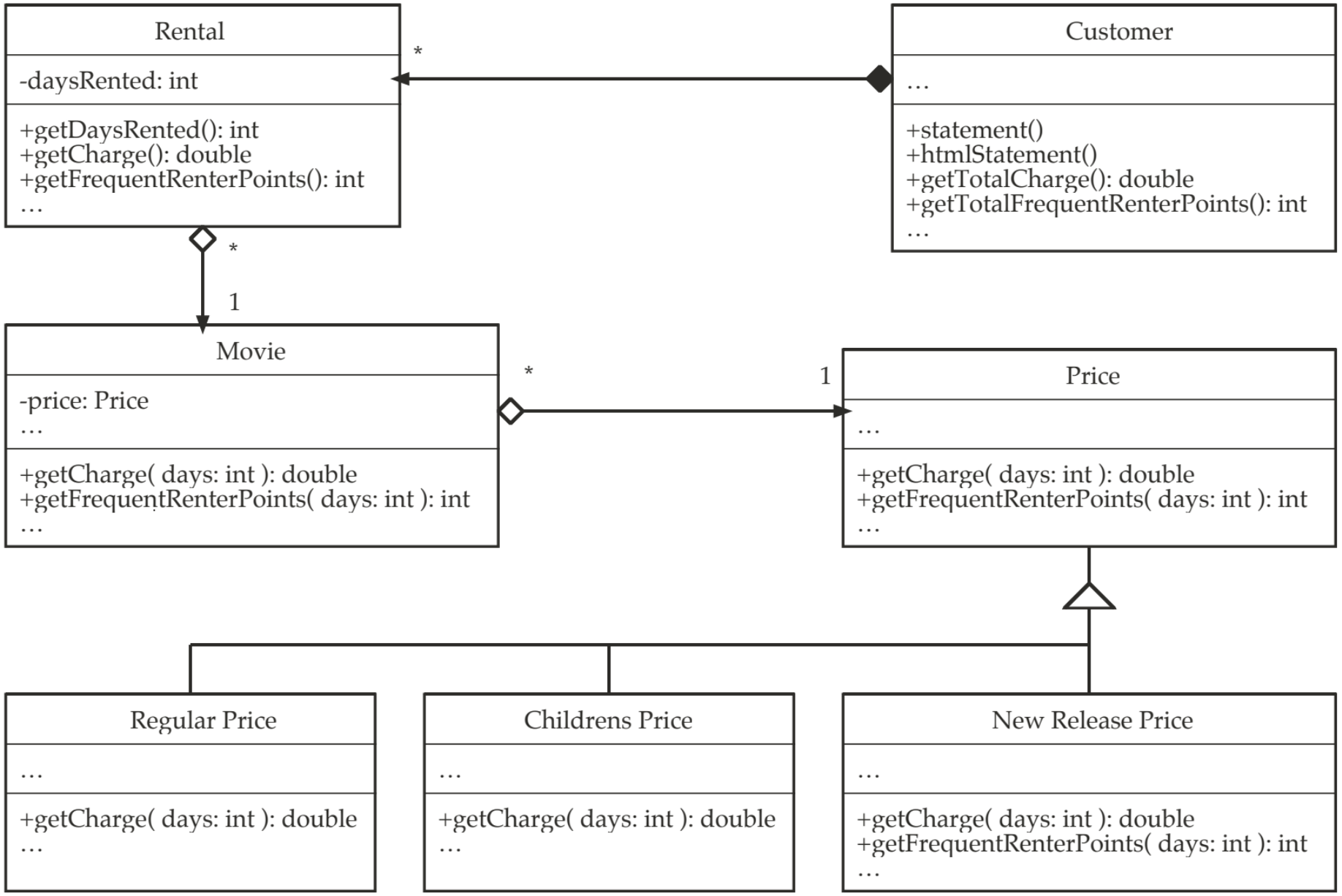
Discussion: More Refactoring?

- How could we further improve the code?

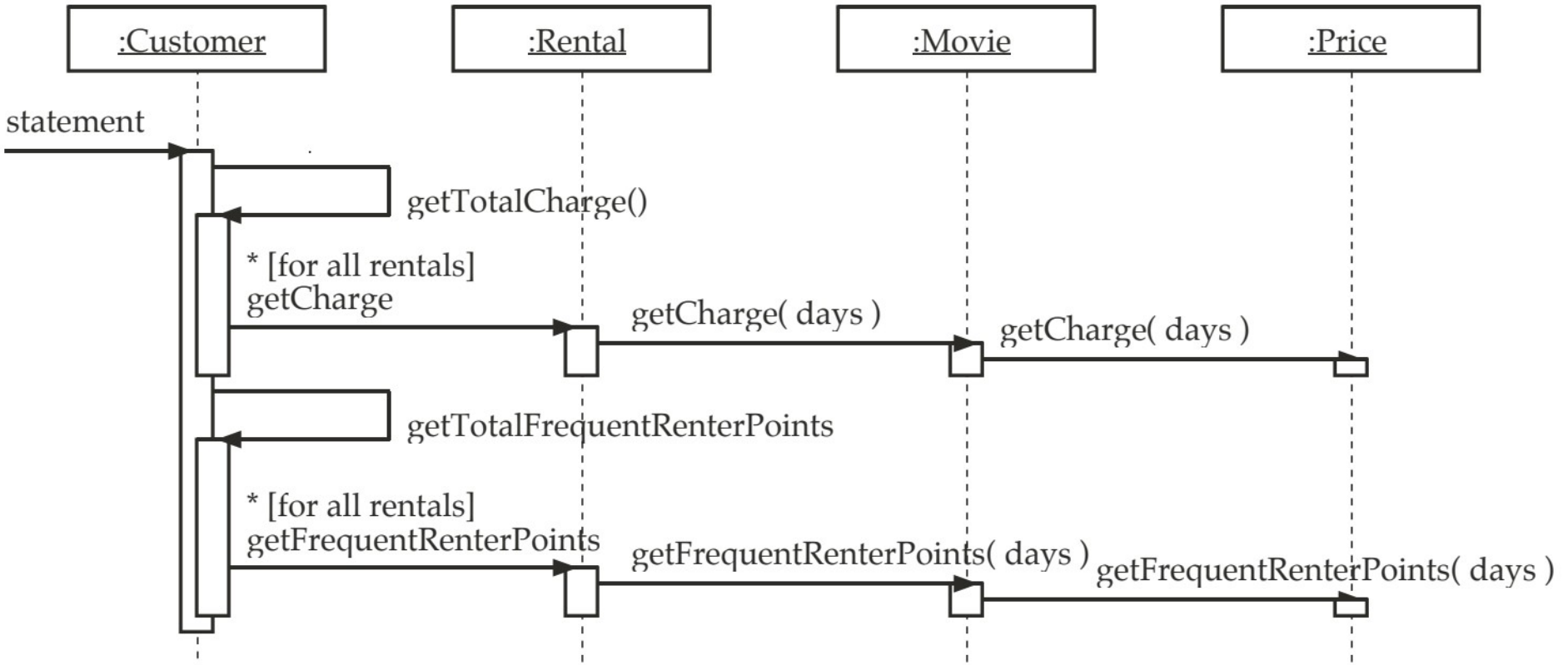
Discussion: More Refactoring?

- How could we further improve the code?
    - Movie types and prices could change in the future, so:
        - `Rental` logic should not depend on *specific* movie types
        - `Price` parent class can be made to make it easier to manage price behavior
            - Relationship with `Movie` class: `Movie` *has a* `Price`

Example Structure Design After Further Refactoring



Example Behavioral Design After Further Refactoring



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

- 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/