Refactoring, Part 2

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 4448/6448 — Lecture 26 — 11/20/08

Introduction

- Credit where Credit is Due
  - Some of the material for this lecture is taken from “Refactoring: Improving the Design of Existing Code” by Martin Fowler; as such some material is copyright © Addison Wesley, 1999
- Last Lecture
  - Refactoring
    - Introduced core ideas
      - Improve design without changing functionality
      - Watch out for “bad smells” in code
    - Covered several examples
- Goals for this Lecture
  - Present a longer, more detailed example

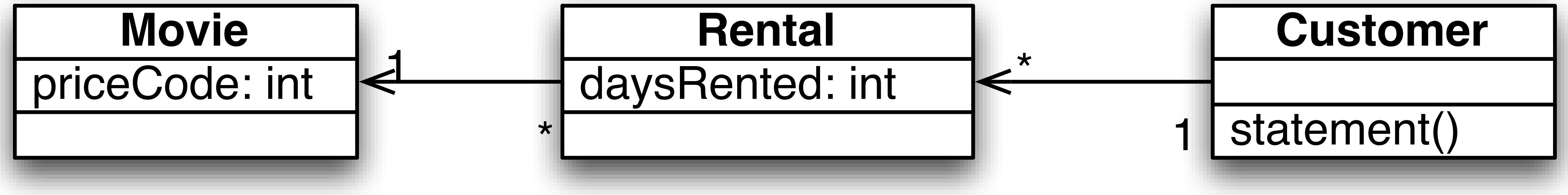
Tutorial

- A simple program for a video store
  - `Movie`
  - `Rental`
  - `Customer`
    - customer object can print a statement (in ASCII)
- We'd like to modify the code to also print a statement in HTML and have discovered that none of the existing code can be reused!
- See example code (available on class website)
  - Added a test case! We will test our code after each refactoring

Does this code need refactoring?

- For such a simple system
  - probably not
- but imagine that these three classes are part of a larger system
  - then the refactorings we do during the tutorial can indeed be useful
    - the point is to imagine following this process on a daily basis in a larger system project
    - refactoring needs to be incremental, systematic, and safe

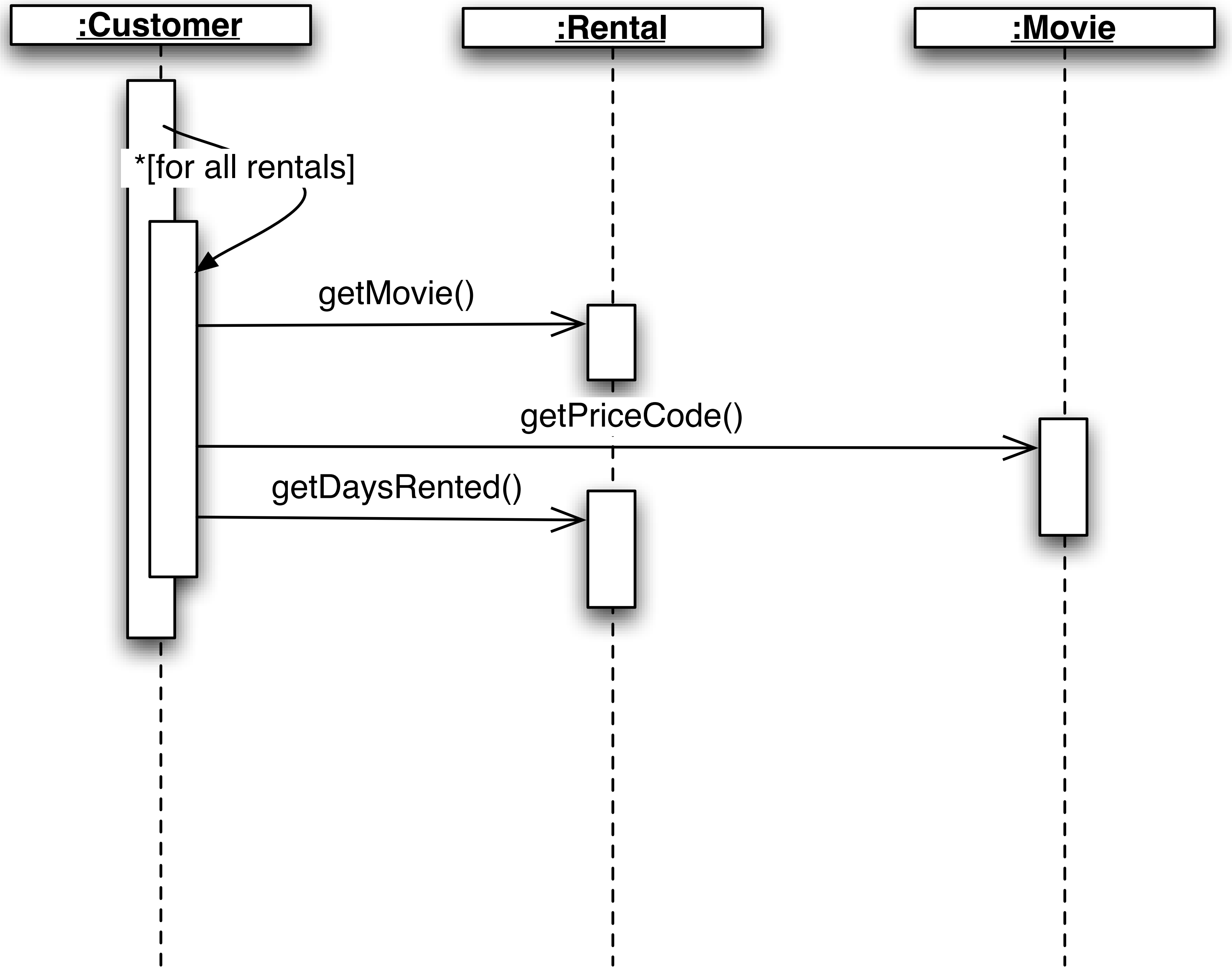
Initial Class Diagram



`statement()` works by looping through all rentals;
for each rental, it retrieves its movie and the number of days it was rented; it also retrieves the price code of the movie

it then calculates the price for each movie rental and the number of frequent renter points and returns the generated statement as a string; (see next slide)

Initial statement() algorithm



Step 1: refactor `statement()`

- Why?
  - It’s a “long method” which is one of the “bad smells” covered last lecture
- Also:
  - our purpose is to add a new method to `Customer` that generates a statement formatted in HTML
  - refactoring `statement()` may lead to code that can be shared with this new function
    - This matches one of Fowler’s conditions for refactoring: cleaning up the code to make it possible to add a new function

How to start?

- We want to decompose the statement() method into smaller pieces
- We'll start with "Extract Method"
  - and target the switch statement first
  - look for local variables: `each` and `thisAmount`
  - `each` is not modified, `thisAmount` is
    - non-modified variables can be passed as parameters (if required)
    - modified variables require more care; since there is only one, we can make it the return value of the new method
- Pitfalls
  - be careful about return types;
    - in the original statement, `thisAmount` is a double
    - but it would be easy to make the mistake of having the new method return an int; if you do, your test will fail because the rounding of ints to doubles would cause some of your amounts to change; try it and see with the Customer class in the step1 directory

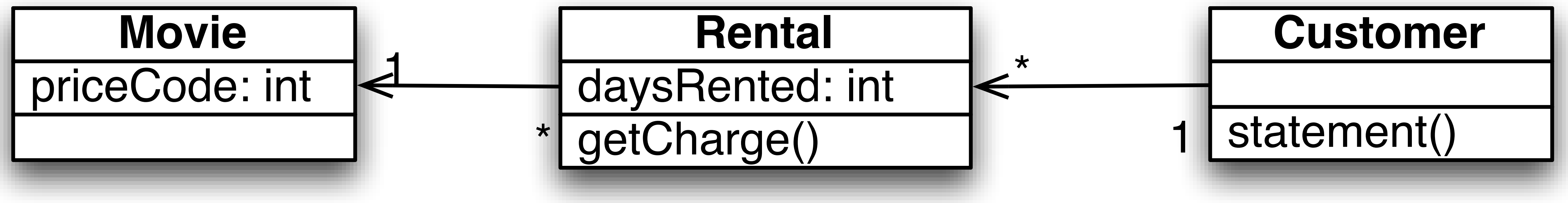
Step 2: rename variables

- The variable names in the new `amountFor()` method don’t make sense, now that they have been moved out of the `statement()` method
    - “Any fool can write code that a computer can understand. Good programmers write code that humans can understand”
- Lets rename them and run our test
    - so far so good!

Step 3: move method

- `amountFor()` uses information from the `Rental` class
  - It does NOT use any information from the `Customer` class
- Methods should be located close to the data they use, so lets move `amountFor()` to the `Rental` class
  - We get rid of a parameter this way
  - Lets also rename the method to `getCharge()` to clarify what it is doing
  - As a result, back in `Customer`, we must delete the old method and change the call to `amountFor(each)` to `each.getCharge()`
- Then we need to compile and test; all good!

New class diagram



No major changes; however Customer is now a smaller class and an operation has been moved to the class that has the data it needs to do its job;

Definitely making progress!

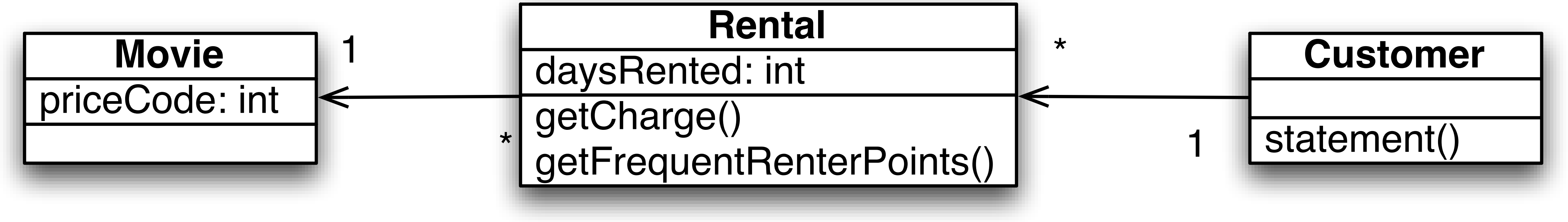
Step 4: Replace Temp with Query

- In the statement() method, `thisAmount` is now redundant. It is set once with the call to `each.getCharge()` and is not changed afterward
    - lets get rid of it.
        - Don't forget to run your test!
- Removing temp variables is a good thing, because they often cause the need for parameters where none are required and can also cause problems in long methods;
    - of course the charge is now calculated twice through the loop, but we can optimize the calculation later (but only if we determine that it is slowing us down)

Step 5: frequent renter points

- Lets do the same thing with the logic to calculate frequent renter points
  - Step 5a: extract method
    - `each` can be a parameter, as in step1
    - `frequentRenterPoints` has a value before the method is invoked, but the new method does not read it; we simply need to use appending assignment outside the method
  - Step 5b: move method
    - Again, we are only using information from `Rental`, not `Customer`, so lets move `getFrequentRenterPoints()` to the `Rental` class
  - Be sure to run your test case after each step

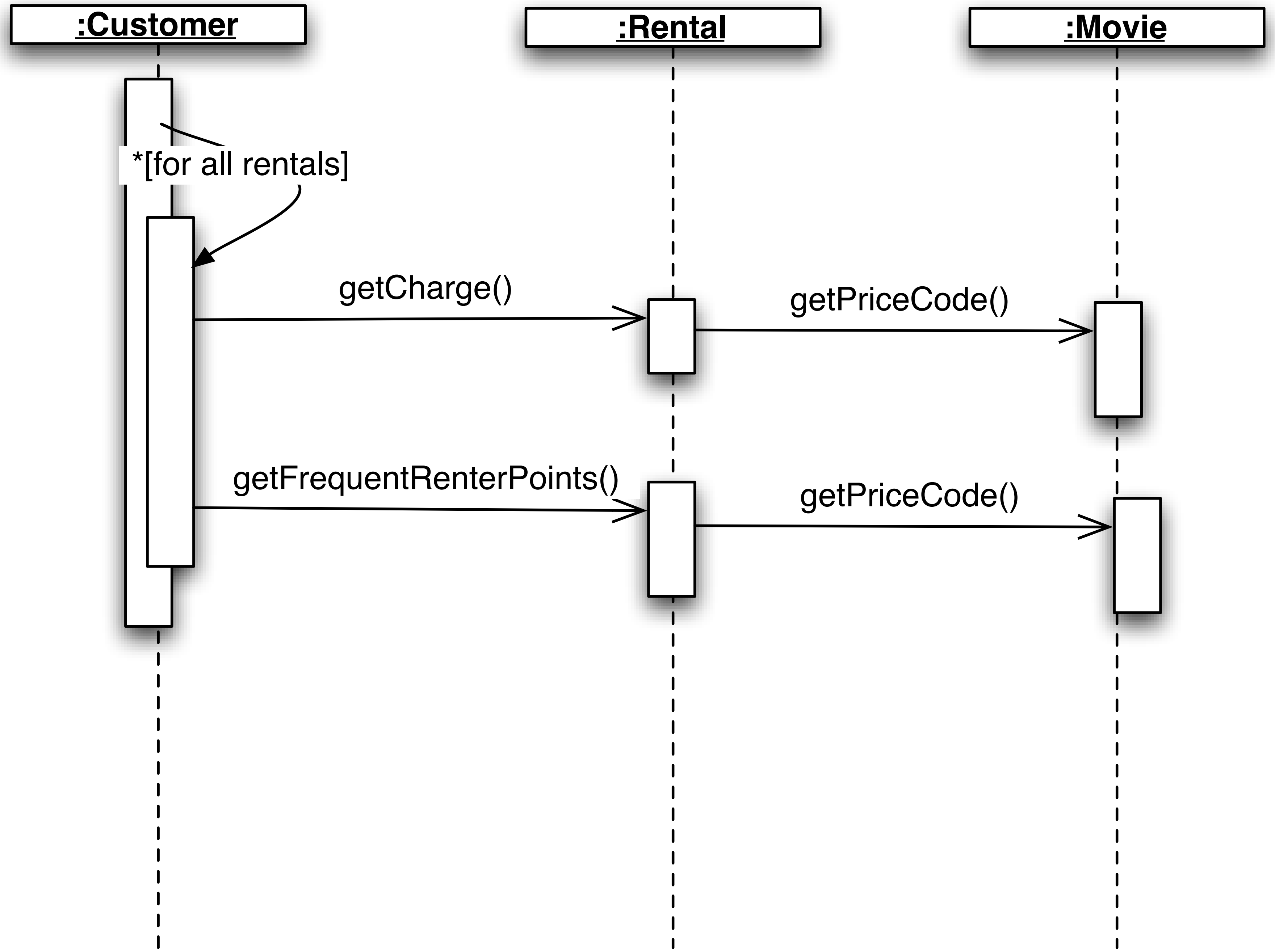
New class diagram



`Customer` continues to get smaller, `Rental` continues to get larger; but `Rental` now has operations that change it from being a “data holder” to a useful object

Our sequence diagram has changed (see next slide); `statement()` used to call the `Movie` class to get the price code for each movie. Now `Rental` takes care of that. And `statement()` now calls methods that have names that mean something rather than presenting lots of code whose purpose may not be clear

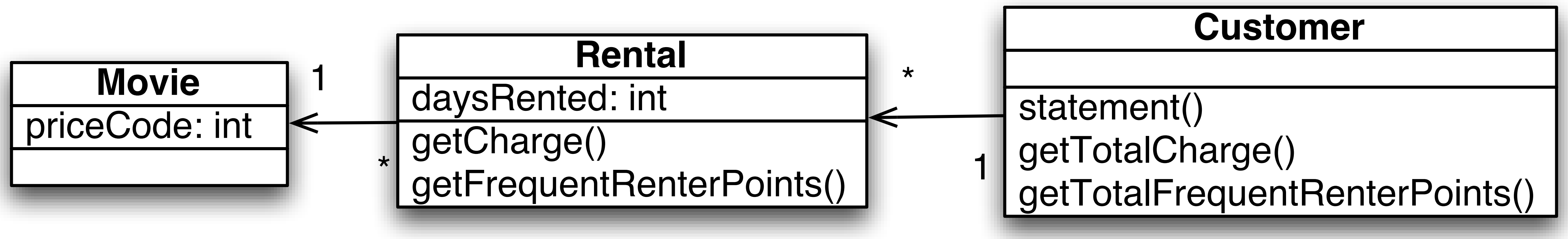
new statement() algorithm



Step 6: Remove temp variables

- `statement()` still has temp variables
    - `totalAmount` and `frequentRentalPoints`
- Both of these values are going to be needed by `statement()` and `htmlStatement()`
    - Lets replace them with query methods
        - little more difficult because they were calculated within a loop; we have to move the loop to the query methods
    - Step 6a: replace `totalAmount`
    - Step 6b: replace `frequentRentalPoints`
        - test after each step

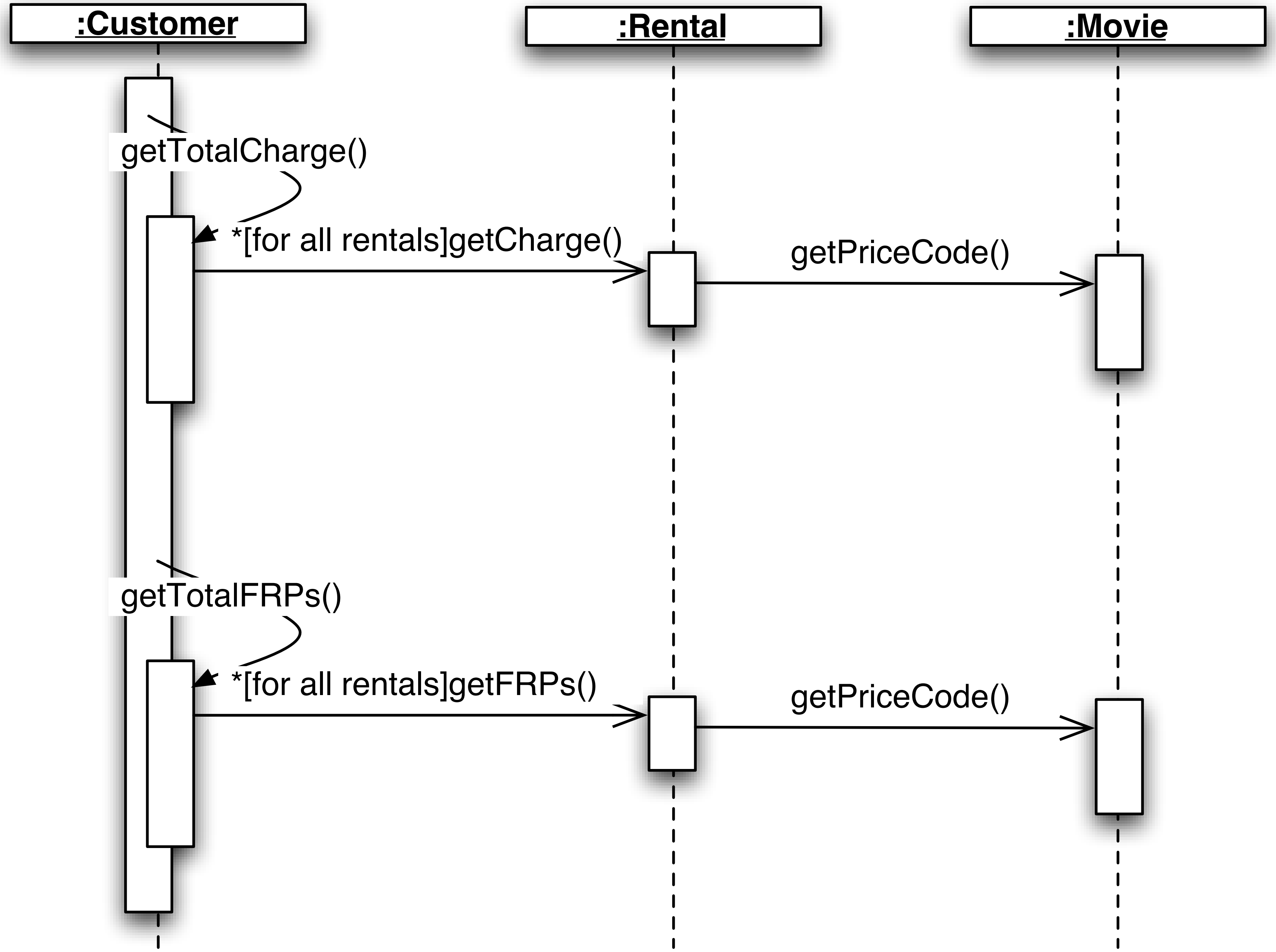
Current class diagram



`Customer` class is now bigger; but has two methods that can be shared between the existing `statement()` method and the planned `htmlStatement()` method

Our sequence diagram has changed again (see next slide) because now we have three loops instead of one; again, performance can be a concern but we should wait until a profiler tells us so!

latest statement() algorithm



Step 7: add `htmlStatement()`

- We are now ready to add the `htmlStatement()` function
  - Note: I'm not going to test this function, but I will add it, so you can see how our refactorings so far, have made it easy to add this function
    - I added a file to the step7 directory that prints out the results of calling `htmlStatement()`; you can send the output to a web browser if you want
  - You can actually improve these two methods using a refactoring called **Form Template Method**, but I will not cover that today
    - You can probably guess how to do it, however, by reviewing the template method design pattern that we discussed in Lecture 22

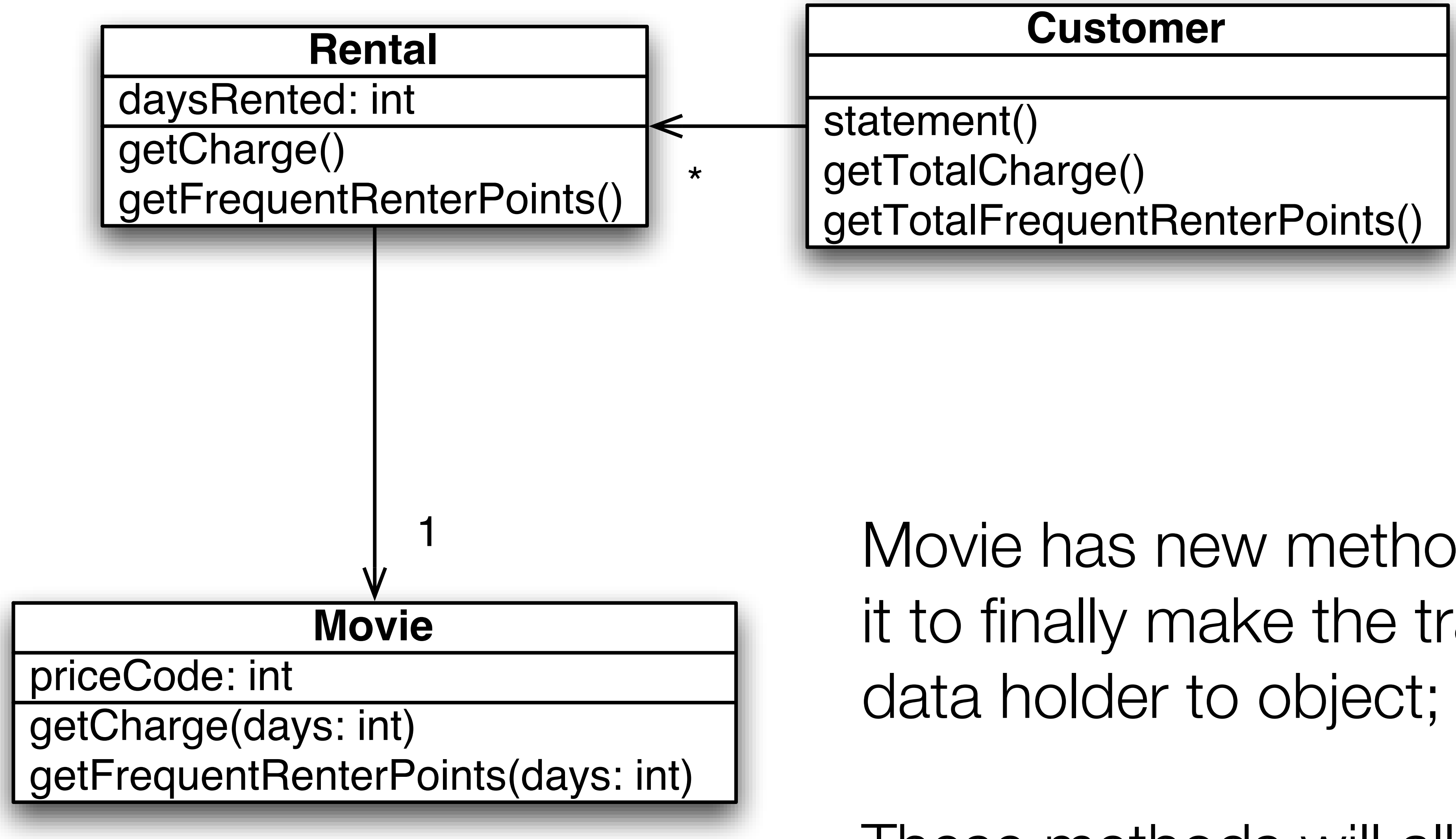
New Requirements

- It is now anticipated that the store is going to have more than the three initial types of movies;
  - as a result of these new classifications, renter points and charges will vary with each new movie type
  - as a result, we should probably move the `getCharge()` and `getFrequentRenterPoints()` methods to the Movie class

Step 8: move methods

- Step 8a: move `getCharge()` to Movie
    - `getCharge()` needs to know the number of days the movie was rented; since this is information that `Rental` has, it needs to be passed as a parameter
- Step 8b: move `getFrequentRenterPoints()` to `Movie`
    - ditto!

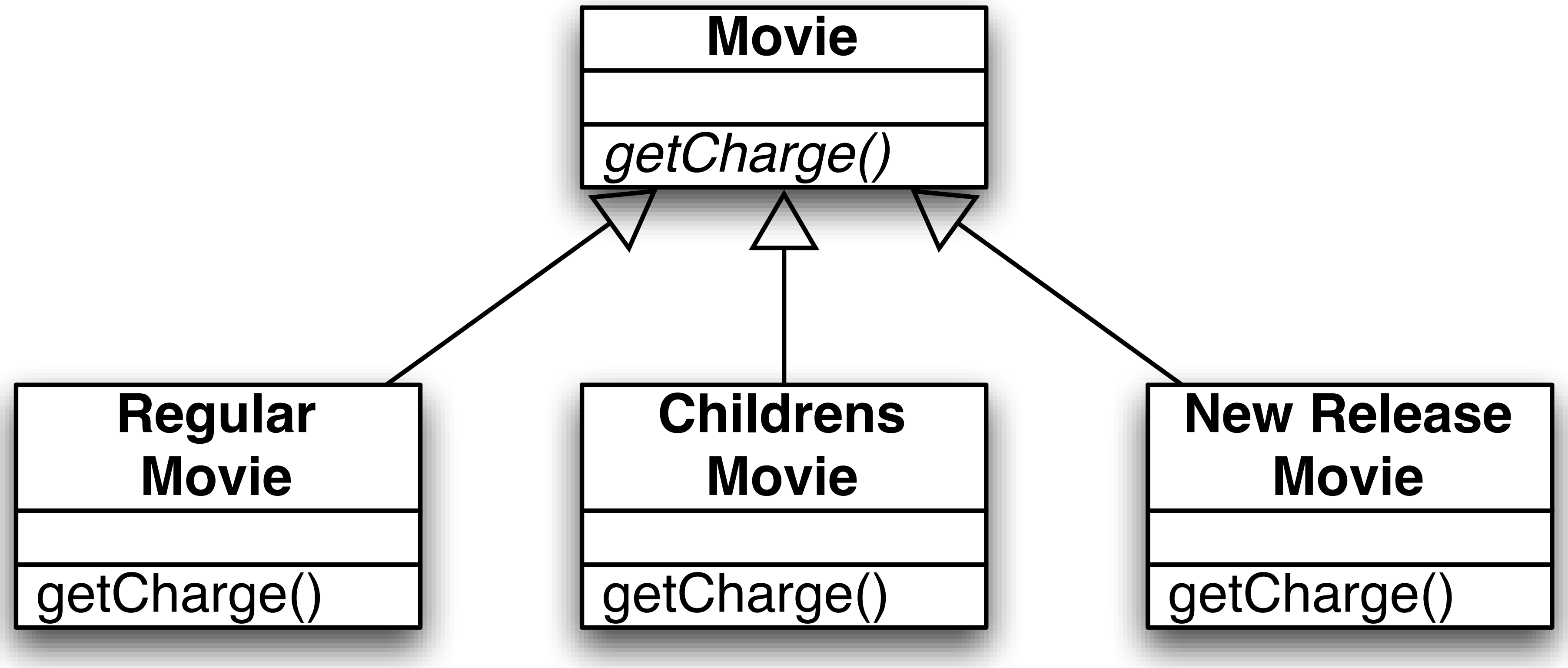
Current class diagram



Movie has new methods, allowing it to finally make the transition from data holder to object;

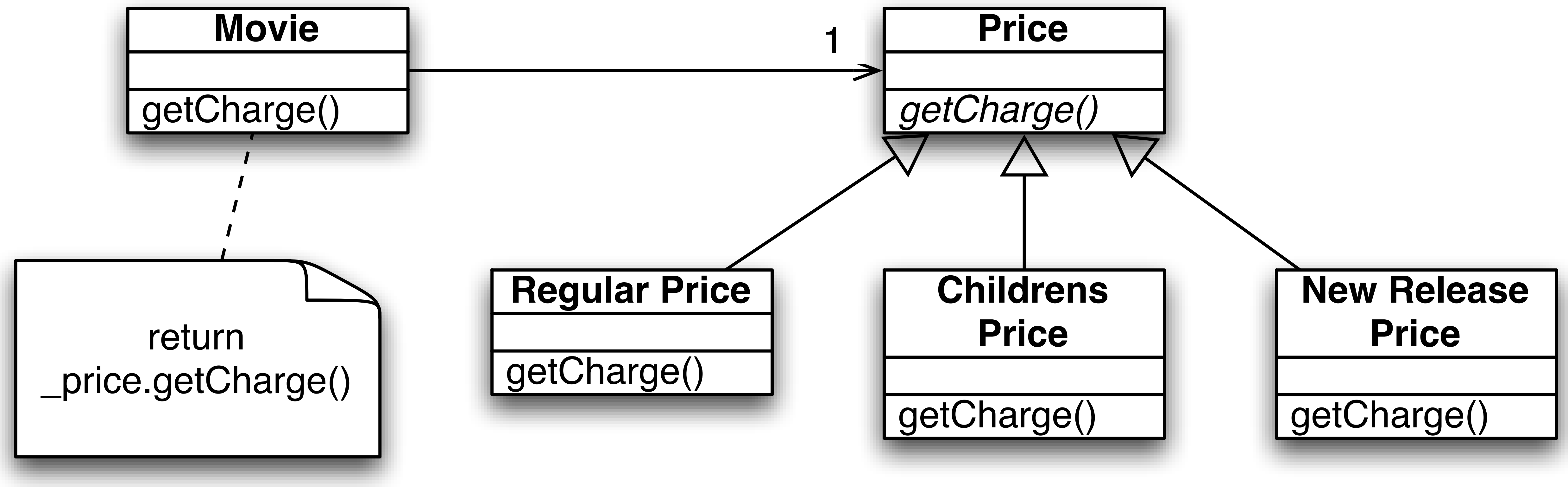
These methods will allow us to handle new types of movie easily

How to handle new Movies?



But movies can change type! A children’s movie when it is first released is a “new release”; later it becomes a children’s movie. So this approach won’t work!

State pattern to the rescue!



A movie has a particular state: its charge (and its renter points) depend on that state; so we can use the state pattern to handle new types of movies (for now, at least)

Step 9: Replace Type Code with State/Strategy

- We need to get rid of our type code (e.g. `Movie.CHILDRENS`) and replace it with a `Price` object
  - We first modify `Movie` to get rid of its `_priceCode` field and replace it with a `_price` object
    - this involves changing the constructor to make use of the `setPriceCode()` method; before it was setting `_priceCode` directly
    - we also have to change `getPriceCode()` and `setPriceCode()` to access the `Price` object
  - (We of course need to create `Price` and its subclasses)

Step 10: Move Method

- Now we need to move the method `getCharge()` to the newly created `Price` class
  - It's a very simple move, we just need to remember to change `Movie` to delegate its `getCharge()` operation to `Price`

Step 11: Replace Conditional with Polymorphism

- Now, we move each branch of the switch statement into the appropriate subclass
    - I do this in one move;
        - Fowler actually recommends moving one branch at a time!
- After you have done the move, change Price's `getCharge()` to an abstract method
- Don't forget to test; everything still works!

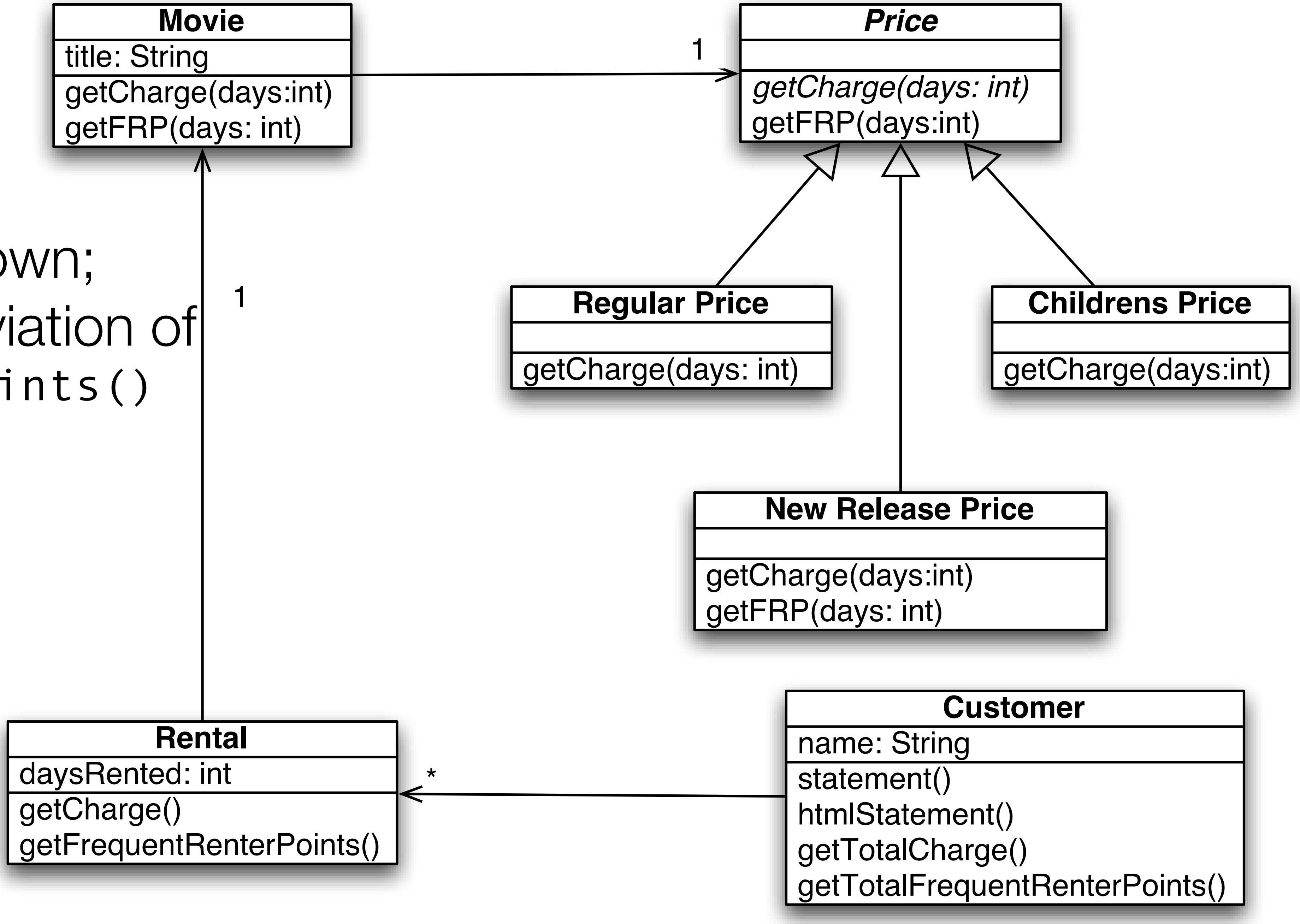
Step 12: handle renter points

- Now we repeat step 10 and 11, this time applying them to frequent renter points
- I combine both steps into one
    - we move the method over to `Price`, and use polymorphism to handle the logic
        - note: this time we leave a default implementation in `Price` and have `NewRelease` override that implementation, since it is the only class that returns a different value
- Run the test and everything still works!

We’re done!

- We’ve added new functionality, changed “data holders” to “objects” and made it very easy to add new types of movies with special charges and frequent rental points
- The final version of the code is in the `after` directory; compile it and run the test: test passed!

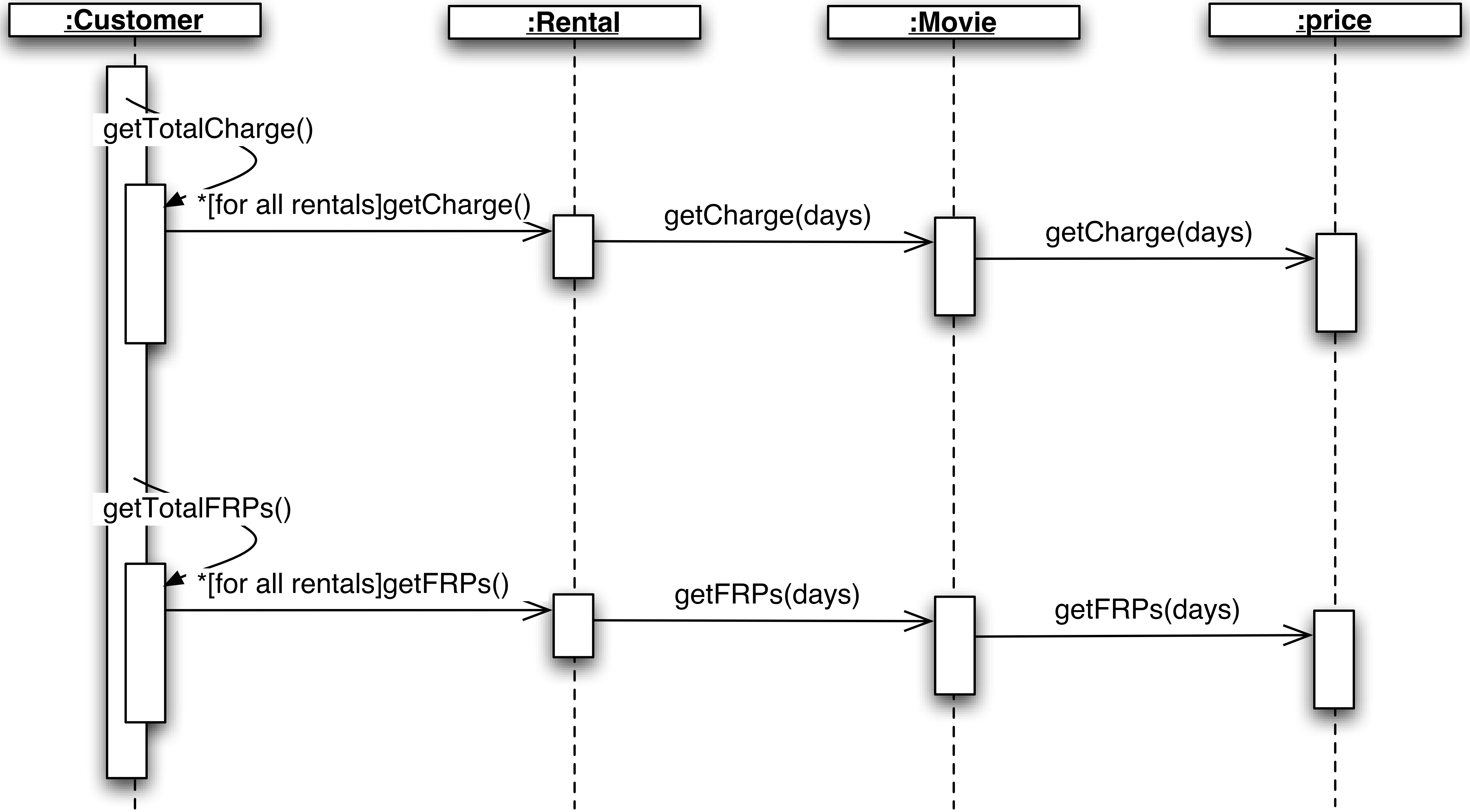
Final class diagram



Note: not all methods and attributes are shown; `getFRP()` is an abbreviation of `getFrequentRenterPoints()`

See next slide for final sequence diagram

Final statement() algorithm



Ken’s Corner: Closures

- Closures: a function that is evaluated in an environment containing one or more bound variables. When called, the function can access these variables.

- Wikipedia: <http://en.wikipedia.org/wiki/Closure_(computer_science)>

- Examples: <http://martinfowler.com/bliki/Closure.html>

Wrapping Up

- Lecture 27: Test Driven Design
- Lecture 28: Additional UML Models and GRASP* Intro
- Lecture 29: GRASP, continued
- Lecture 30: Concurrency in OO Systems

* General Responsibility Assignment Software Patterns