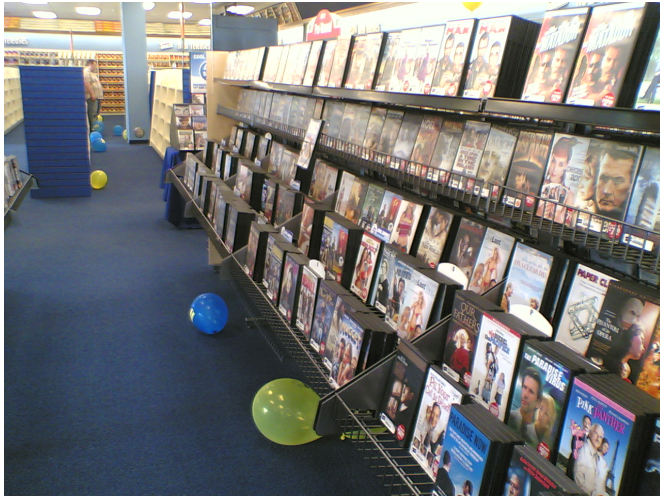


# Refactoring Exercise



Maria Grazia Pia

*INFN Genova, Italy*

Maria.Grazia.Pia@cern.ch

<http://www.ge.infn.it/geant4/training/APC2016/>

# Exercise: The Video Store



**Grab basic concepts**  
**Get into the habit of refactoring**

M. Fowler, Refactoring, Addison-Wesley Professional, 1999, Chapter 1  
*(translated from Java into C++)*

# Setting up the computing environment

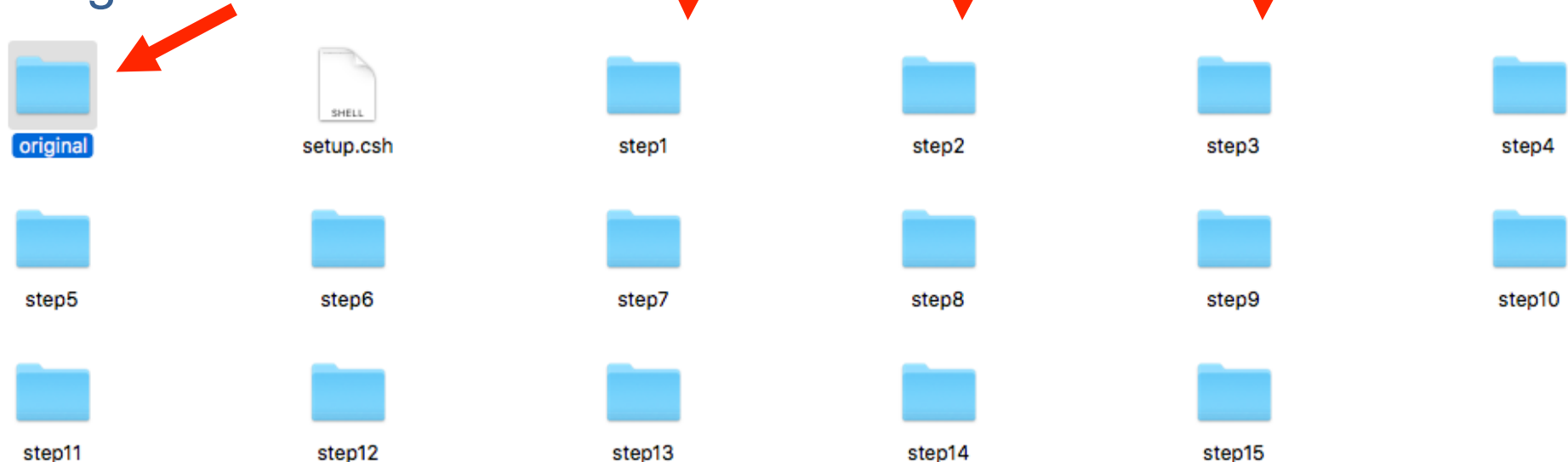
- Details at <https://www.ge.infn.it/geant4/training/APC2016/>
- The exercise has been tested on Scientific Linux 6 with gcc 4.9.3 and on Mac OS 10.11 (El Capitan) with Xcode 7.2.1
  - Presumably it would work smoothly on other platforms too
- The following tools should be installed
  - **cmake**, available from <https://cmake.org/>
  - **googletest**, available from <https://github.com/google/googletest>
  - **git** (*not strictly necessary*)
- Quick instructions for googletest installation  
<https://www.ge.infn.it/geant4/training/APC2016/gtest.html>
- Define an environment variable corresponding to googletest path  
setenv **GTESTPATH** *\$HOME/Documents/g4dev/googletest/gtest\_install*
- Follow the guidance for the exercise at  
<https://www.ge.infn.it/geant4/training/APC2016/exercise.html>

# Download the source code for the exercise

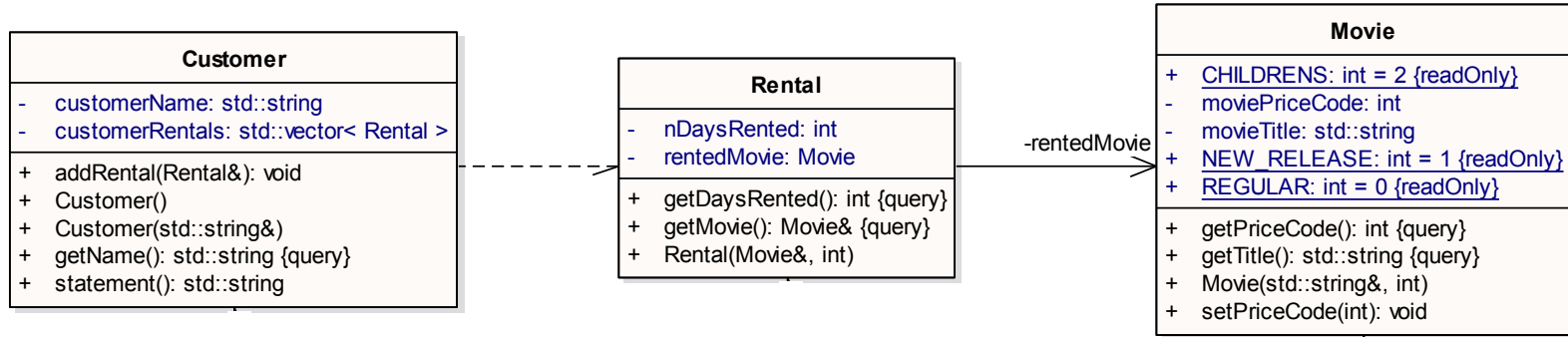
- Download a copy of the exercise source code from <https://github.com/mariagraziapia/VideoStoreAPC/releases/tag/v2.0>
- Unpack the source code and go into the source directory
  - `unzip VideoStoreAPC-v.2.0.zip`
  - `cd VideoStoreAPC-v.2.0`

15 steps of the refactoring process

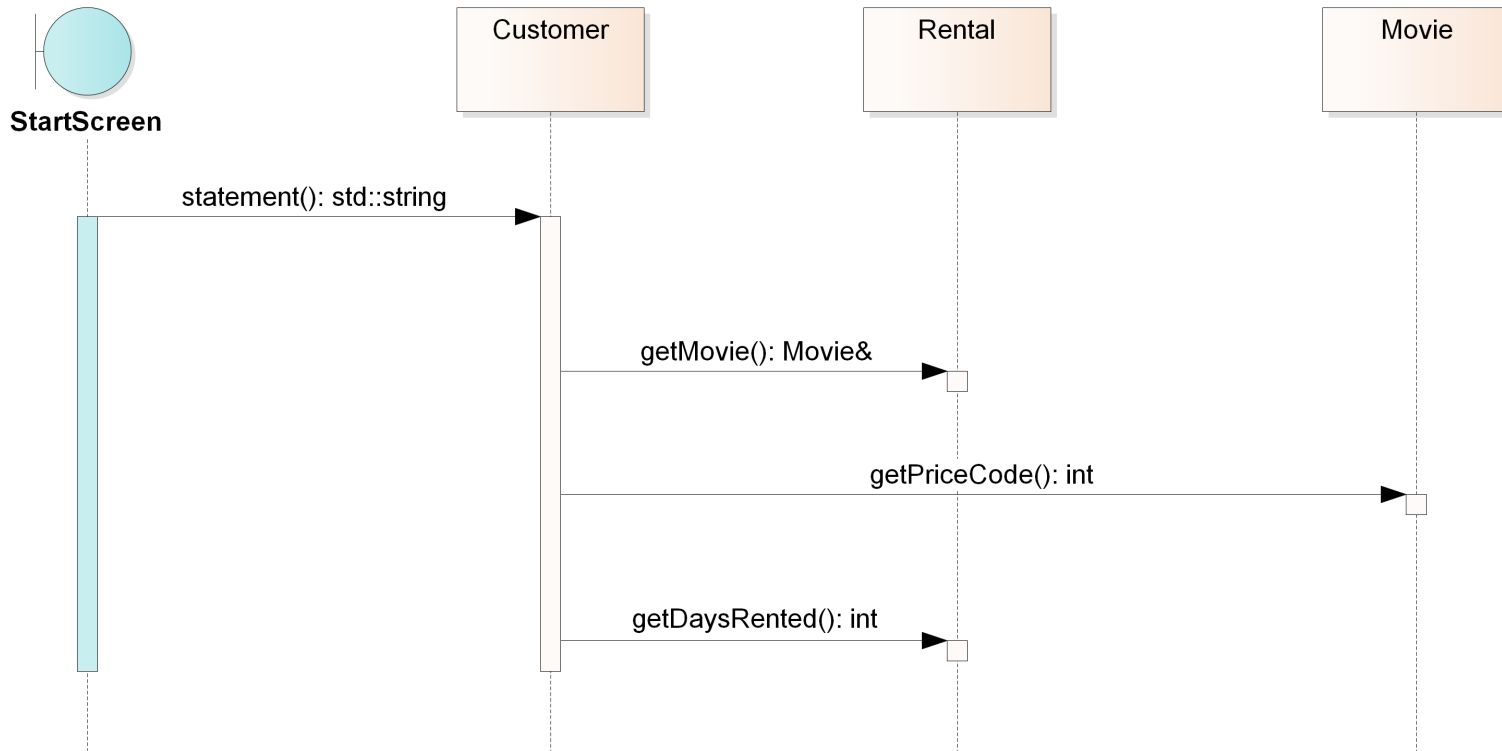
Original code



# Original code



## Original code



# A first look at the code

## ● Not well designed and certainly not object oriented

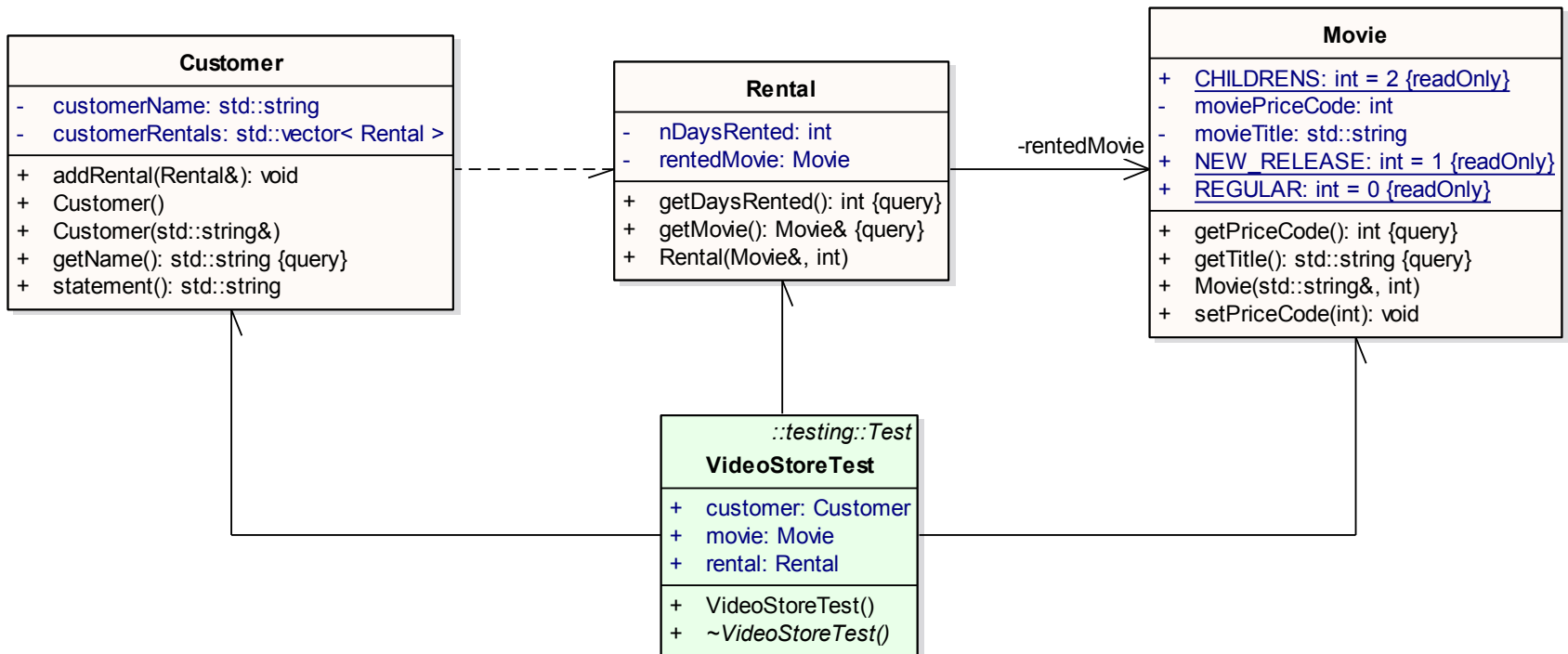
- The long routine in the Customer class does far too much
- Many of the things that it does should really be done by the other classes

## ● Difficult to change

- Suppose that they *want a statement printed in HTML*
- It is impossible to reuse any of the behavior of the current statement method for an HTML statement.
- One would end up with writing a whole new method that duplicates much of the behavior of statement.
- But what happens when the charging rules change? You have to fix both statement and htmlStatement and ensure the fixes are consistent.
- The users want to make changes to the way they classify movies, but they haven't yet decided on the change they are going to make.
- The statement method is where the changes have to be made to deal with changes in classification and charging rules

# Step 0: Tests

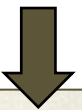
- One needs a solid set of **tests** for that section of code.
  - Risk of introducing bugs while modifying the code
- Tests must be **self-checking**
  - They either say "OK"
  - or they print a list of failures



# Running the unit tests

- Build testVideoStore.cc
- `cd original`
- `make` *the test is automatically run*

The output should look like



Build and run the tests in the same way at each step

`make clean`  
to remove \*.o and executable

To rerun the tests:  
`./testVideoStore`

```
[=====] Running 7 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 7 tests from VideoStoreTest
[ RUN     ] VideoStoreTest.testgetPriceCode
[         OK ] VideoStoreTest.testgetPriceCode (0 ms)
[ RUN     ] VideoStoreTest.testsetPriceCode
[         OK ] VideoStoreTest.testsetPriceCode (0 ms)
[ RUN     ] VideoStoreTest.testgetTitle
[         OK ] VideoStoreTest.testgetTitle (0 ms)
[ RUN     ] VideoStoreTest.testgetDaysRented
[         OK ] VideoStoreTest.testgetDaysRented (0 ms)
[ RUN     ] VideoStoreTest.testgetMovie
[         OK ] VideoStoreTest.testgetMovie (0 ms)
[ RUN     ] VideoStoreTest.testgetName
[         OK ] VideoStoreTest.testgetName (0 ms)
[ RUN     ] VideoStoreTest.teststatement
[         OK ] VideoStoreTest.teststatement (0 ms)
[-----] 7 tests from VideoStoreTest (0 ms total)

[-----] Global test environment tear-down
[=====] 7 tests from 1 test case ran. (0 ms total)
[ PASSED ] 7 tests.
```

# Refactoring in 15 steps

For each step N:

- Refactor the code in VideoStoreAPC-v.2.0/stepN-1/
- The solution is in VideoStoreAPC-v.2.0/stepN/
- **Try to do the suggested refactoring yourself**
- following the guidance in <http://www.ge.infn.it/geant4/training/APC201/exercise.html>
- Whenever you modify the code, run the tests

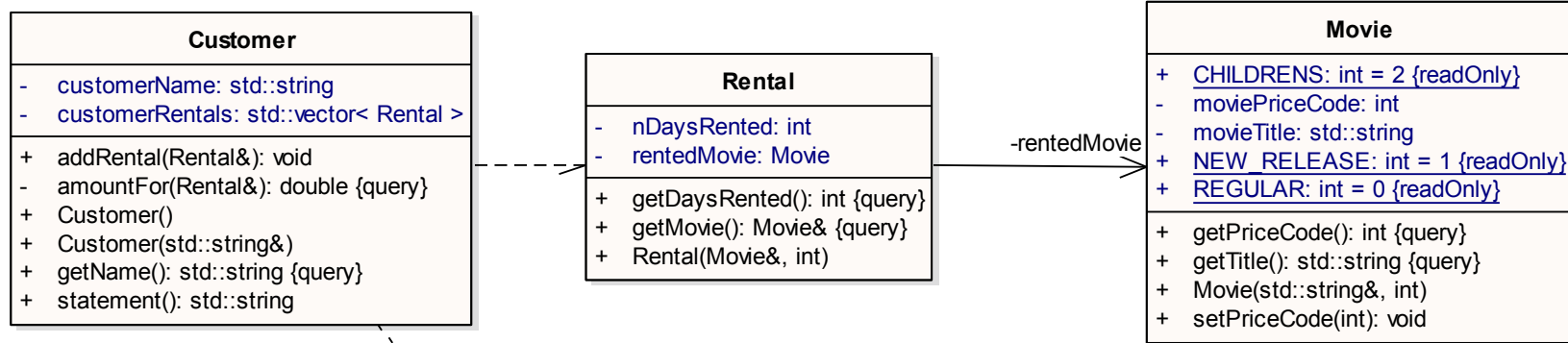
# Step 1: Extract Method

Bad smell:

the long method in Customer  
Decompose it in small pieces

## Step 1: Extract Method

M. Fowler, Refactoring - Chapter 1: Refactoring, a First Example



Extract Method:  
ammountFor

The first phase of refactorings in this exercise shows how to split up the long method and move the pieces to better classes

Find a logical clump of code and use **Extract Method**

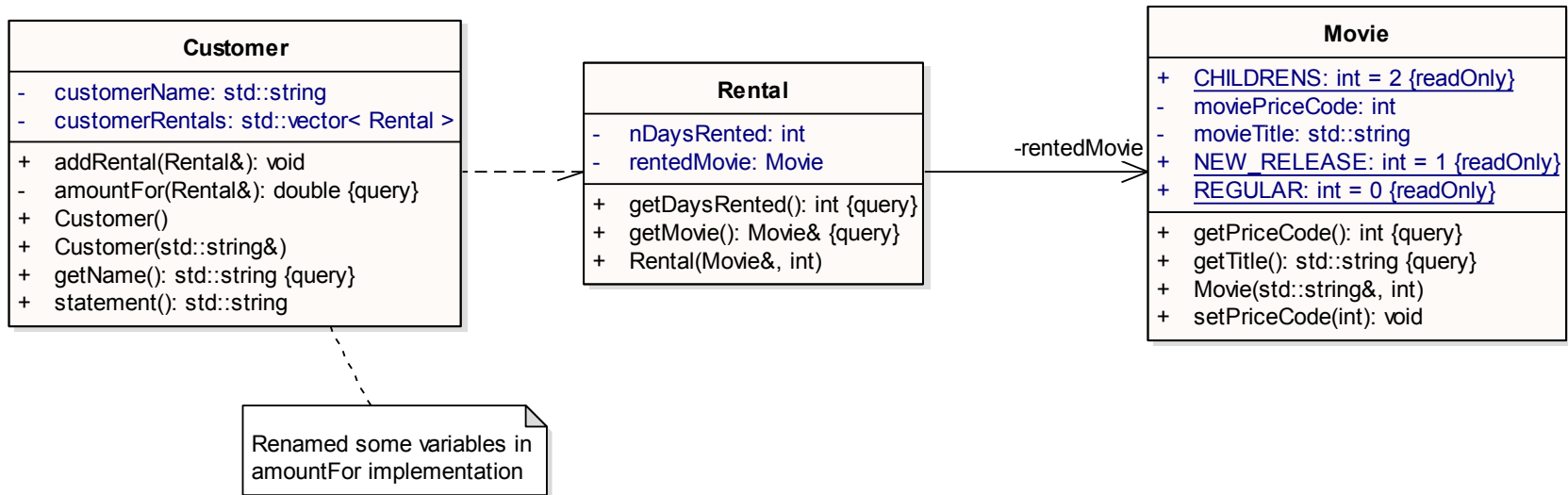
Candidate: **switch** statement to extract into its own method

Solution in step2/

# Step 2: Renaming Variables

## Step 2

No software design change



No design change

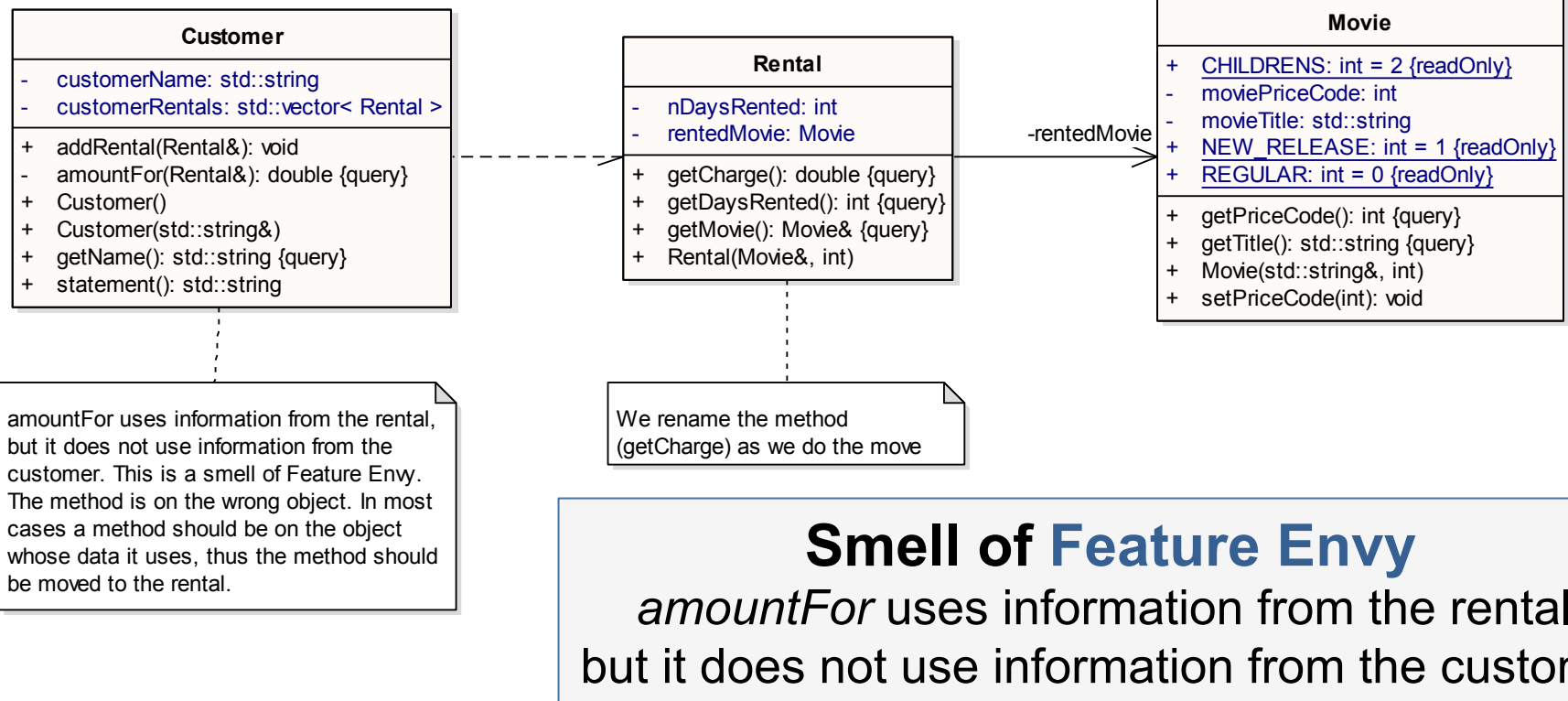
Some of the variable names in amountFor could be **better renamed**

Is renaming worth the effort? Absolutely. Good code should communicate what it is doing clearly, and variable names are a key to clear code.

Solution in step3/

# Step 3: Move Method

## Step 3 - Move Method



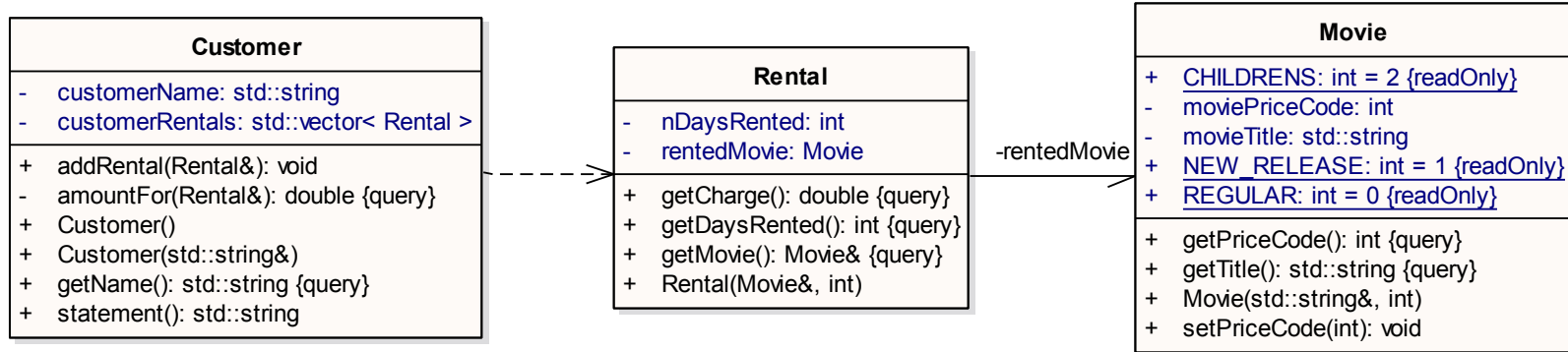
In most cases a method should be on the object whose data it uses:  
the method should be moved to the rental

- Use **Move Method**
- Rename the method (*getCharge*) as we do the move
- Replace the body of *Customer::amountFor* to delegate to the new method

# Step 4: Replace Temp with Query

## Step 4 - Replace Temp with Query

No change to the software design



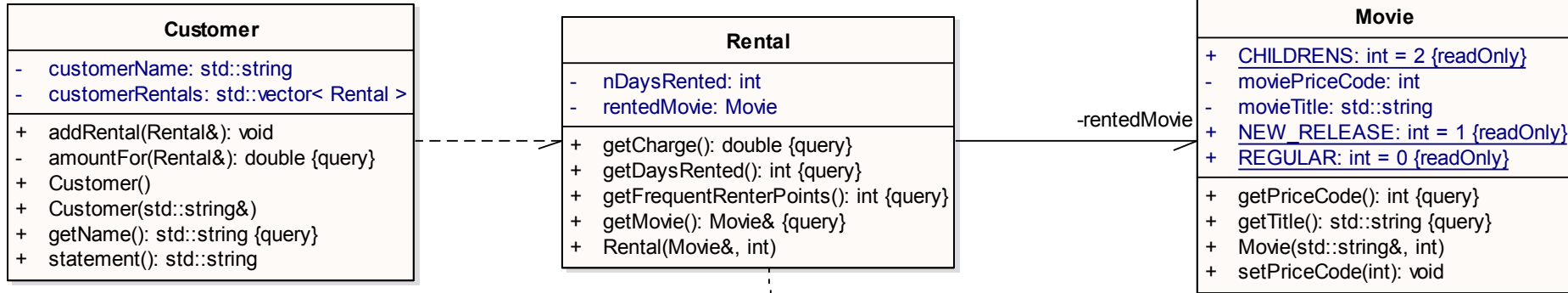
*thisAmount* is now redundant.

It is set to the result of *each.getCharge* and not changed afterward

Thus we can eliminate *thisAmount* by using  
**Replace Temp with Query**

# Step 5: Extracting Frequent Renter Points

## Step 5 - Extracting Frequent Renter Points

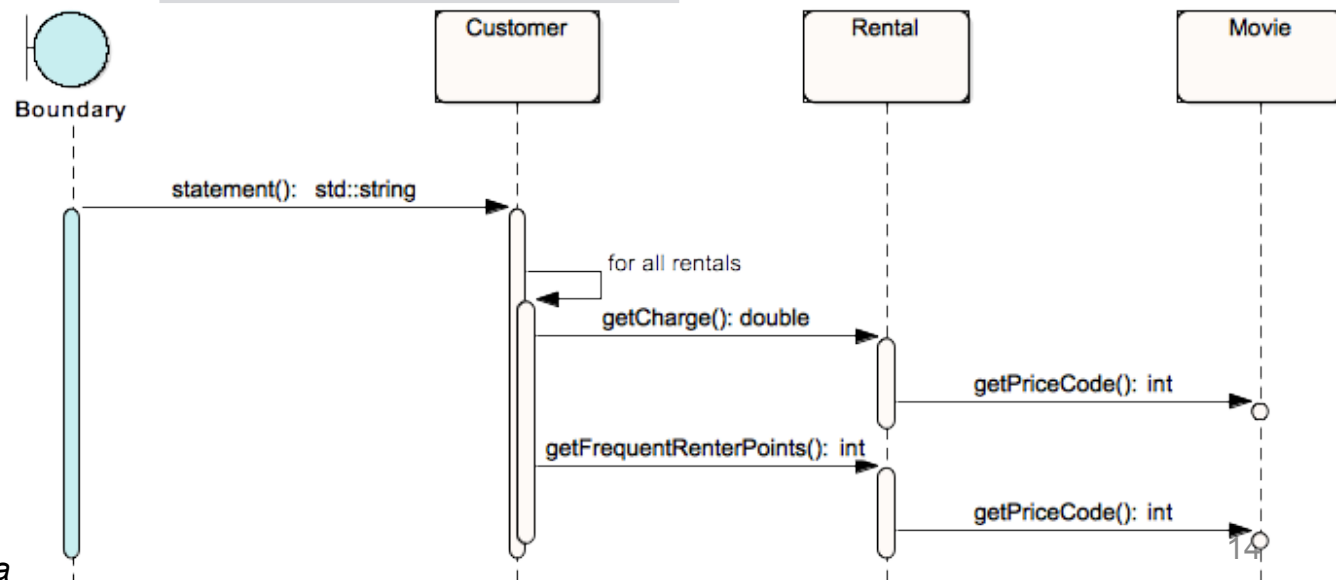


## Step 5 - Sequence

We add the getFrequentRenterPoints method (similar to getCharge in step 3).

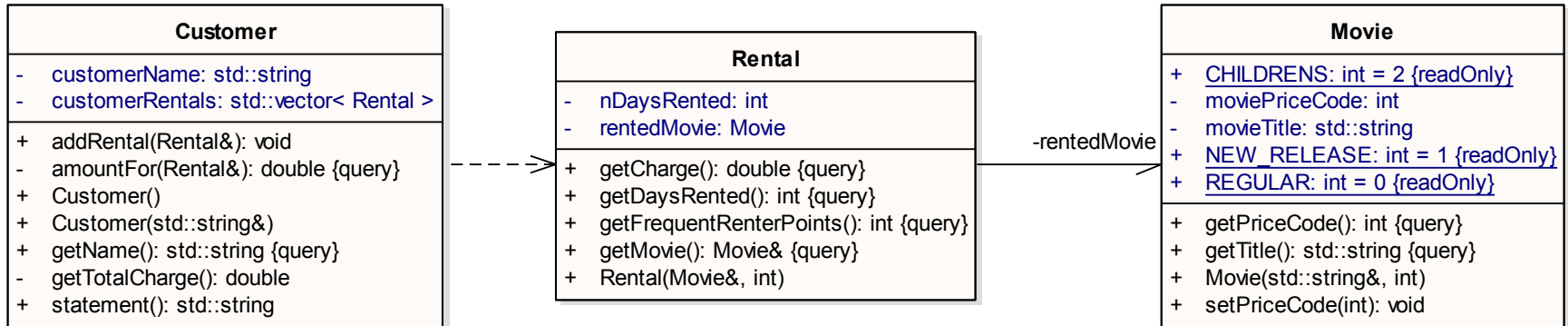
We do a similar thing for the frequent renter points.

**Extract Method**  
on the frequent renter points part of the code



# Step 6: Removing Temps

## Step 6 - Removing Temps



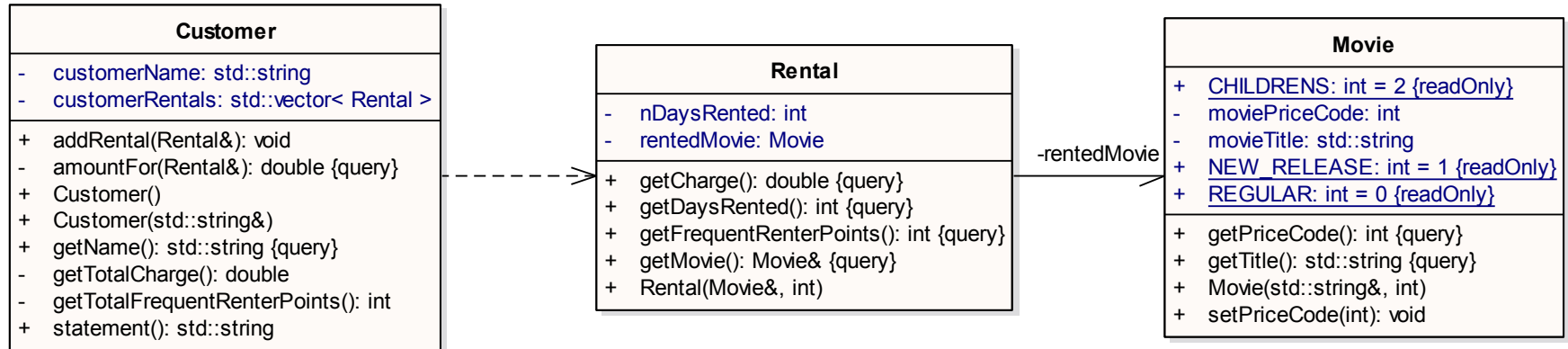
We add the getTotalCharge method.

In this case we have two temporary variables, both of which are being used to get a total from the rentals attached to the customer. Both the ASCII and HTML versions require these totals.

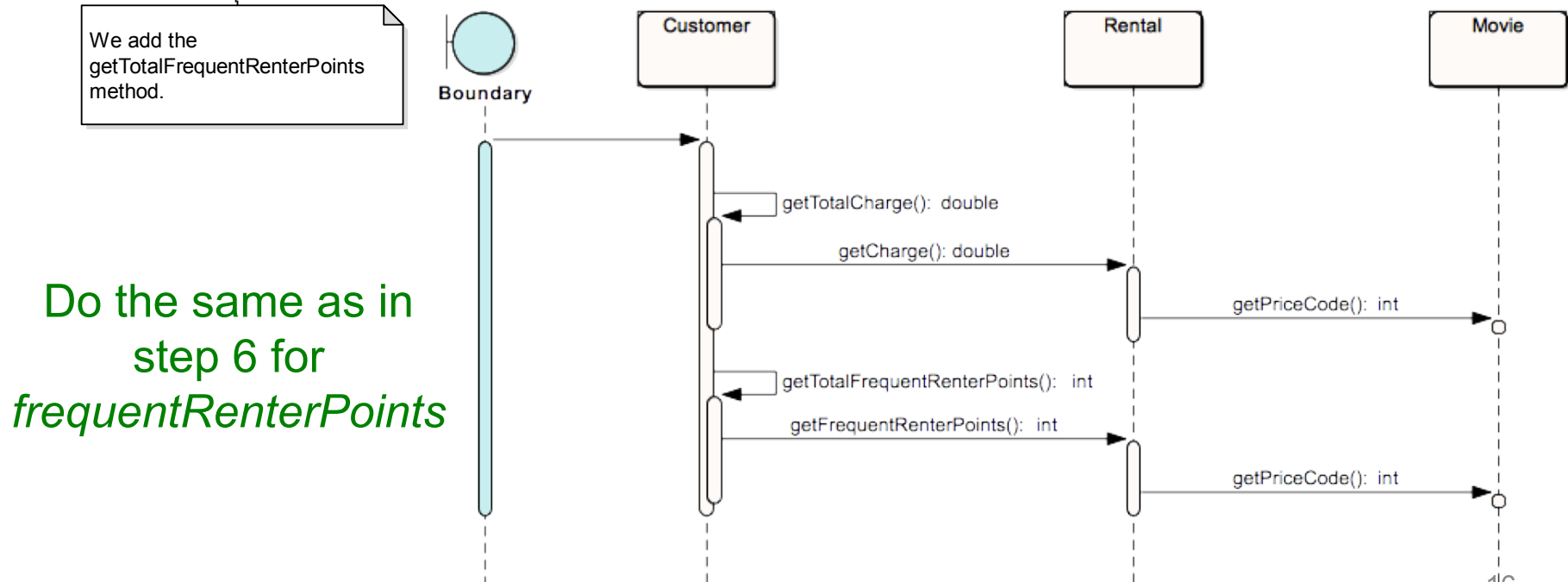
**Use Replace Temp with Query** to replace *totalAmount* and *frequentRenterPoints* with query methods

Queries are accessible to any method in the class, thus encourage a cleaner design  
We begin by replacing *totalAmount* with a *charge* method on *customer*

# Step 7: Still about removing temps

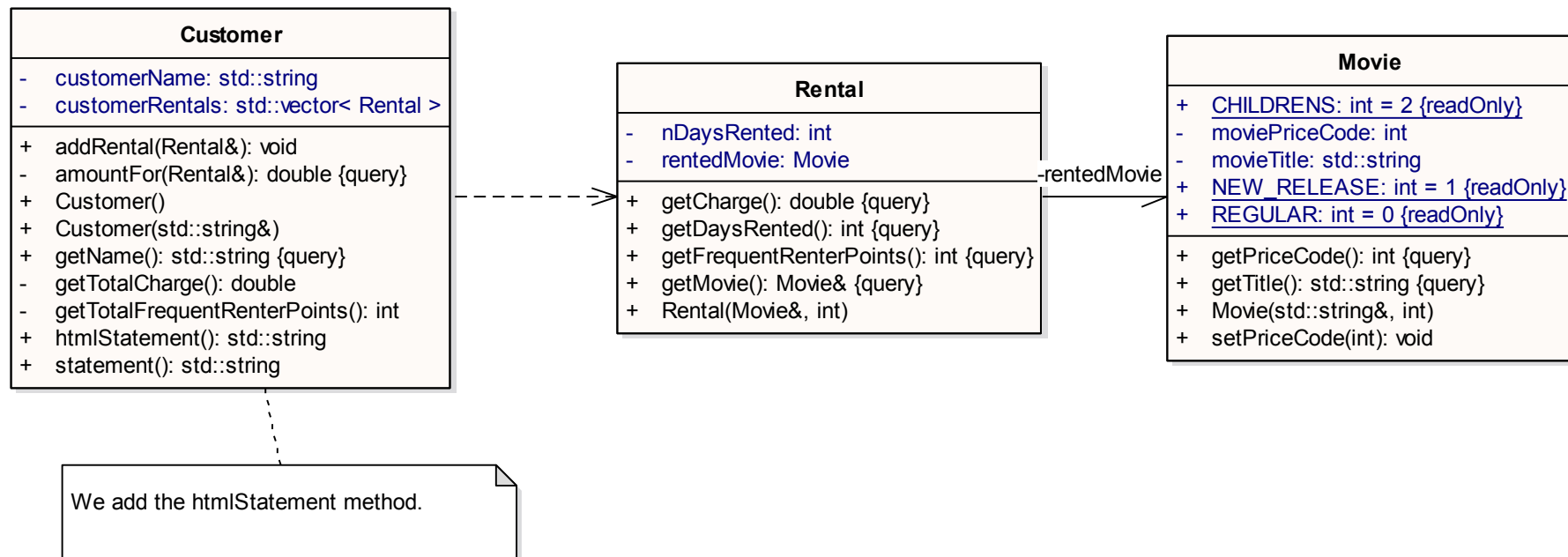


Step 7 - Sequence diagram after extraction of totals



# Step 8: Adding new functionality

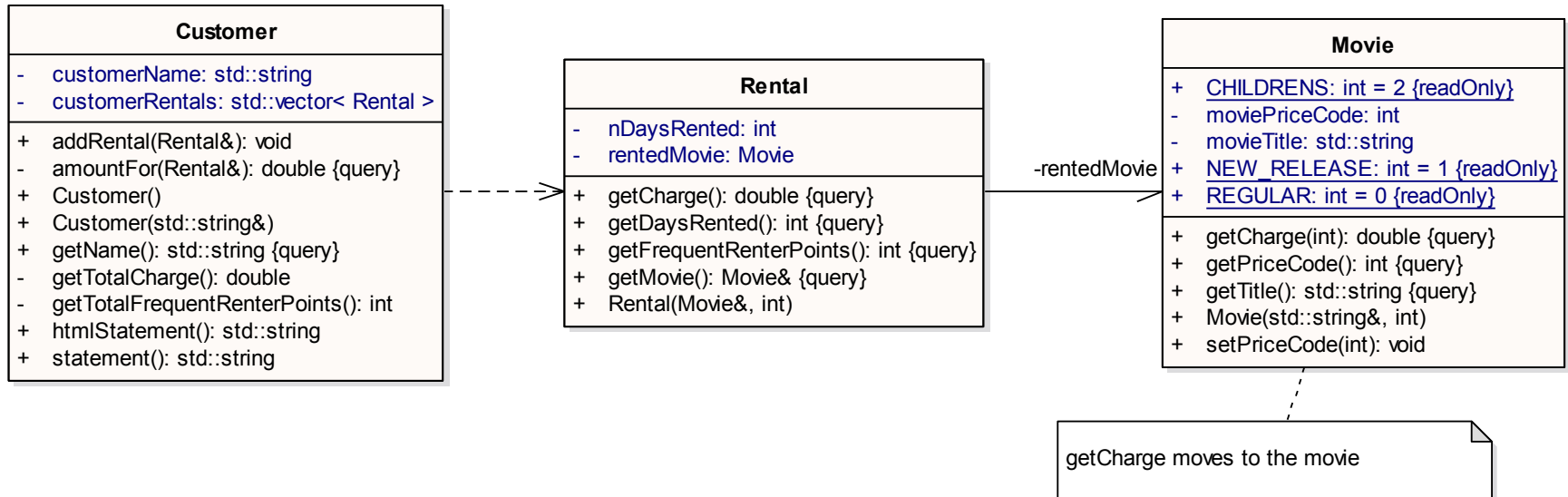
## Step 8 - Adding new functionality



Add htmlStatement() to Customer

# Step 9: Replacing the Conditional Logic on Price Code with Polymorphism

## Step 9 - Replacing the Conditional Logic on Price Code with Polymorphism



The first part of this problem is that switch statement.

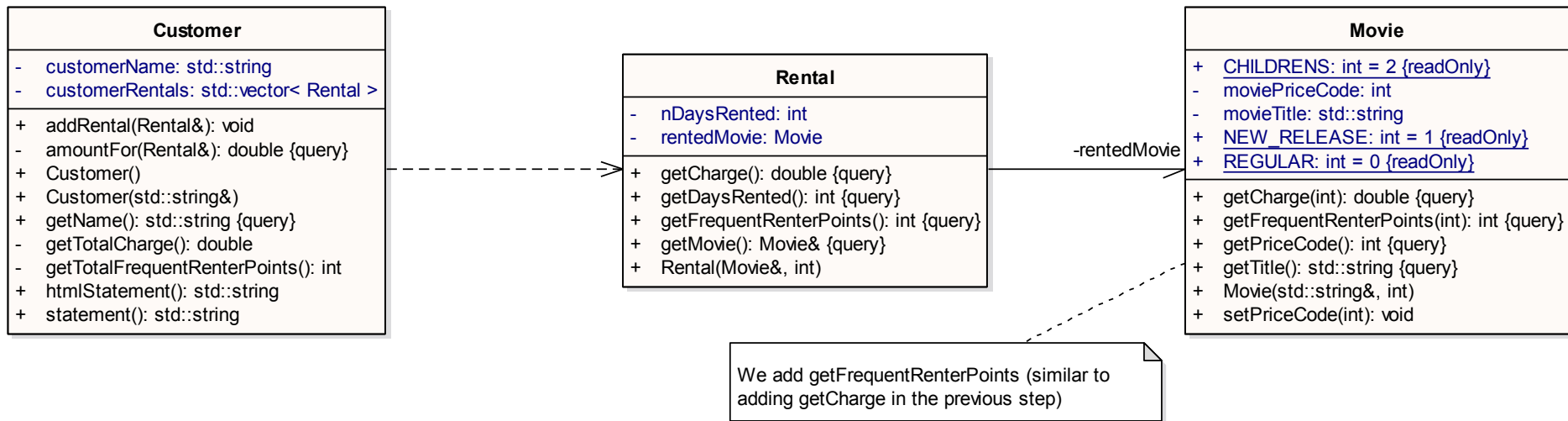
It is a bad idea to do a switch based on an attribute of another object.

If you must use a switch statement, it should be on your own data, not on someone else's

This implies that `getCharge` should move onto movie

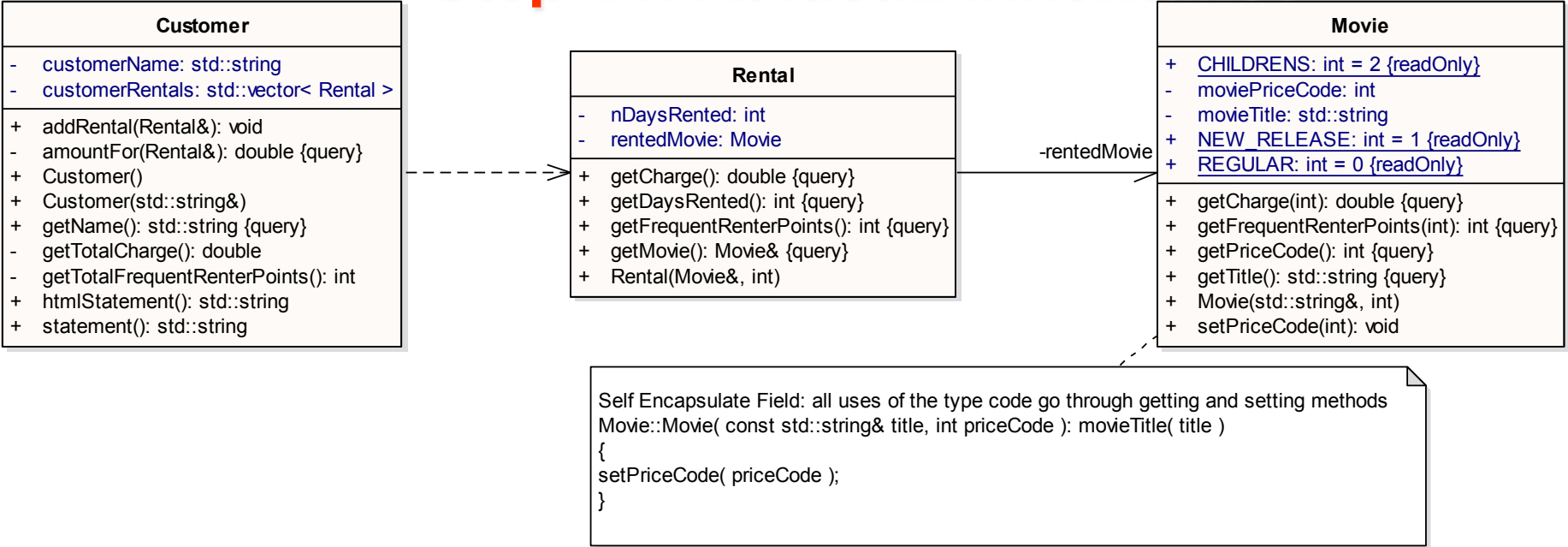
# 10: Still about replacing the Conditional Logic with Polymorphism

## Step 10 - Still about replacing the Conditional Logic with Polymorphism



Do the same as in step 9  
with the frequent renter point calculation

# Step 11: At last... Inheritance



Replace the *switch* statement by using **polymorphism**

We have several types of movie that have different ways of answering the same question

This sounds like a job for **subclasses**

A movie can change its classification during its lifetime

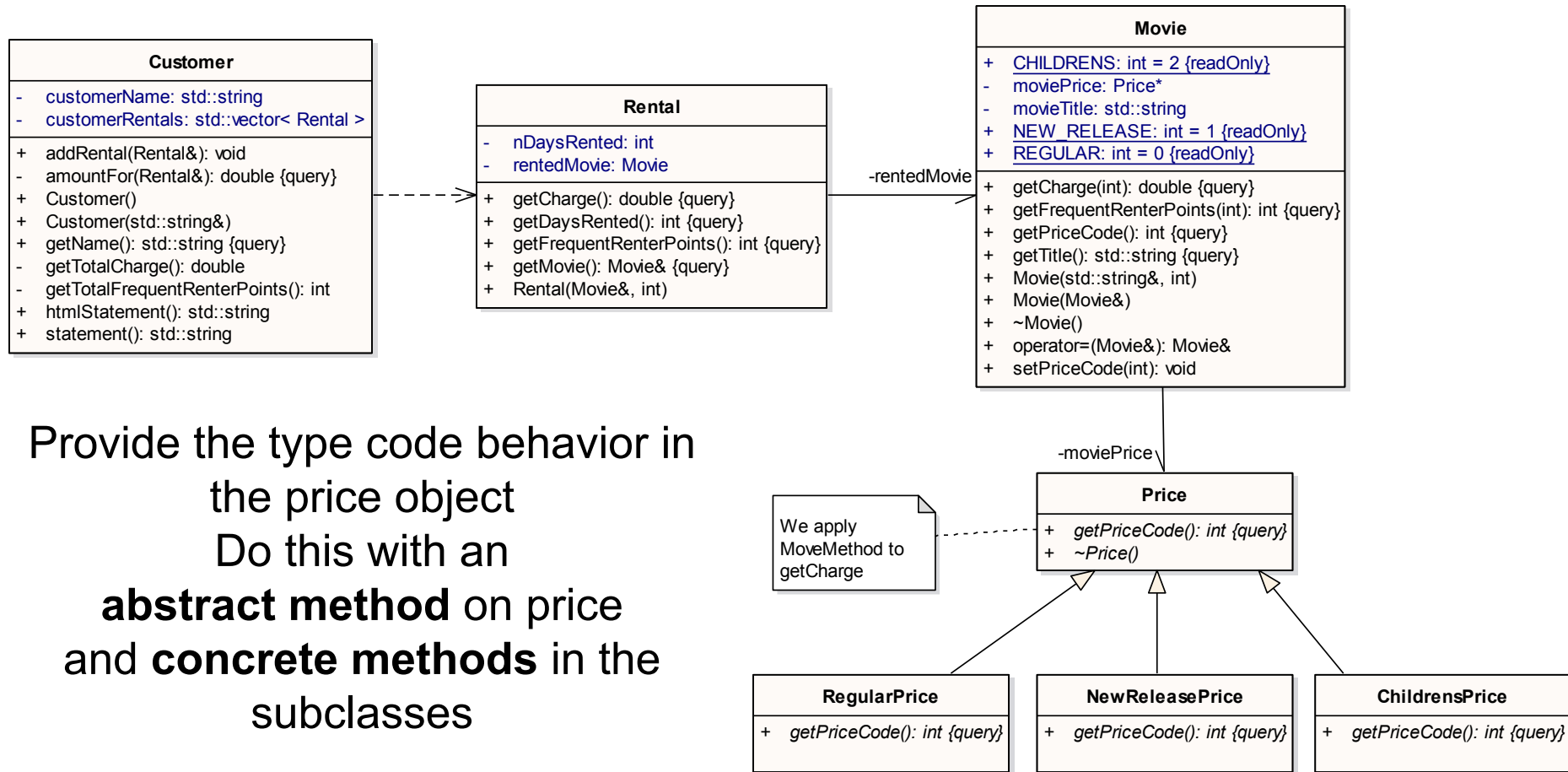
An object cannot change its class during its lifetime

Solution: **use the State pattern**

1. **Replace Type Code** with **State/Strategy: Self Encapsulate Field** on the type code
2. **Move Method** to move the switch statement into the price class
3. **Replace Conditional with Polymorphism** to eliminate the switch statement

# Step 12: Adding new classes

## Step 12 - Adding new classes



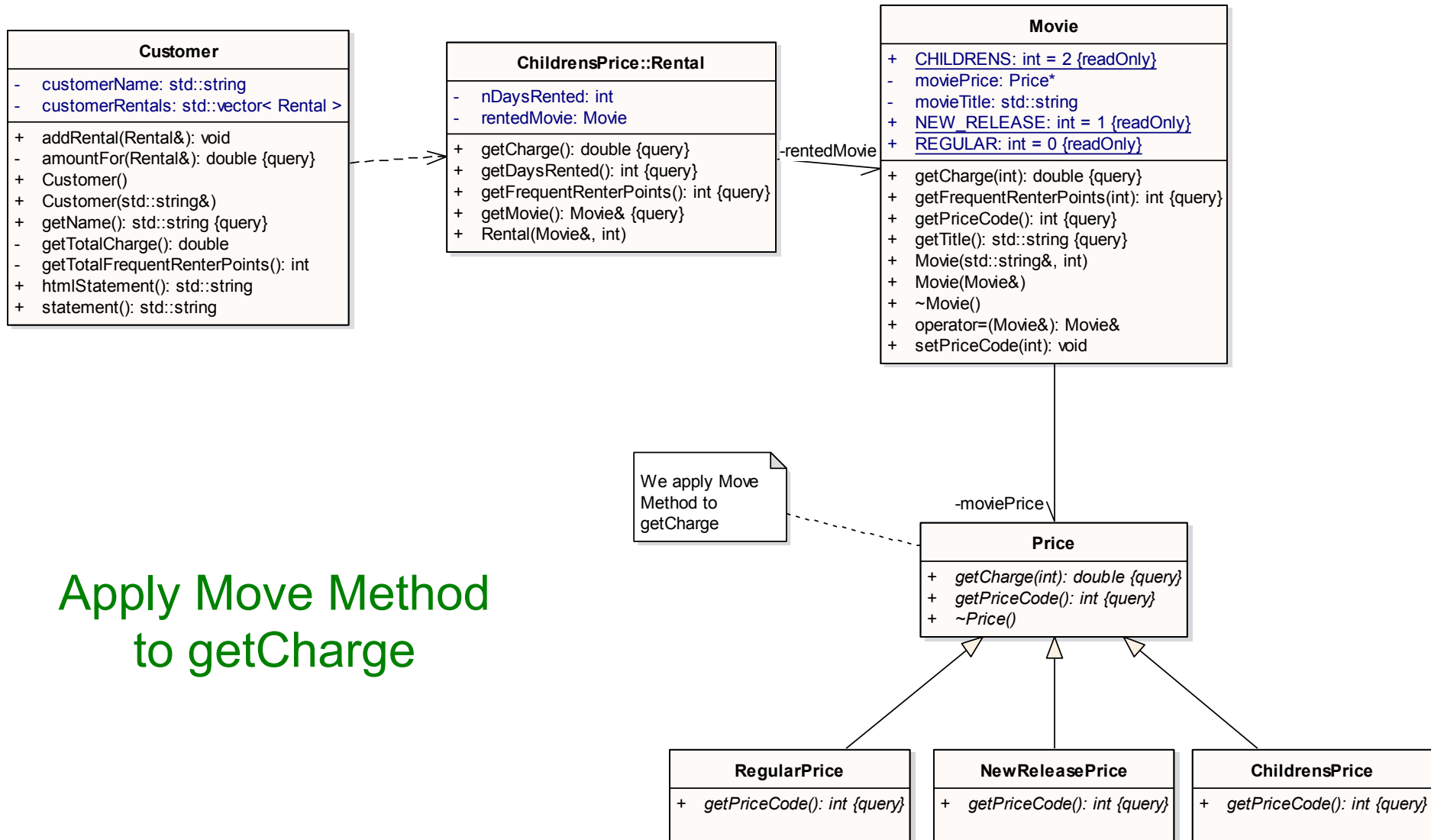
Provide the type code behavior in the price object  
Do this with an **abstract method** on price and **concrete methods** in the subclasses

Change the movie's accessors for the price code to use the new class

Add the new classes

# Step 13: Move Method

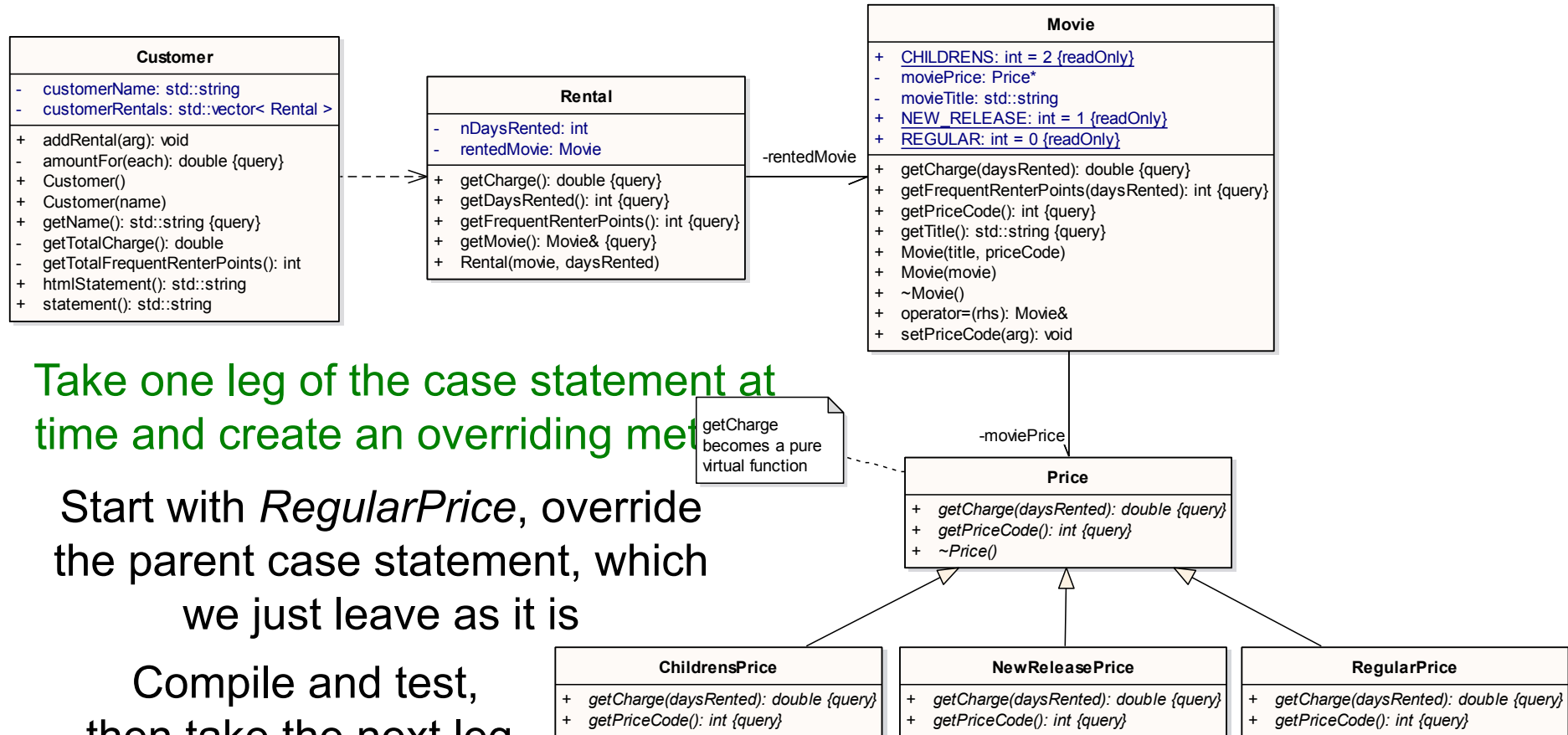
## Step 13 - Move Method



Apply Move Method  
to getCharge

# Step 14: Replace Conditional with Polymorphism

## Step 14: Replace Conditional with Polymorphism



Take one leg of the case statement at time and create an overriding method

Start with *RegularPrice*, override the parent case statement, which we just leave as it is

Compile and test, then take the next leg, compile and test

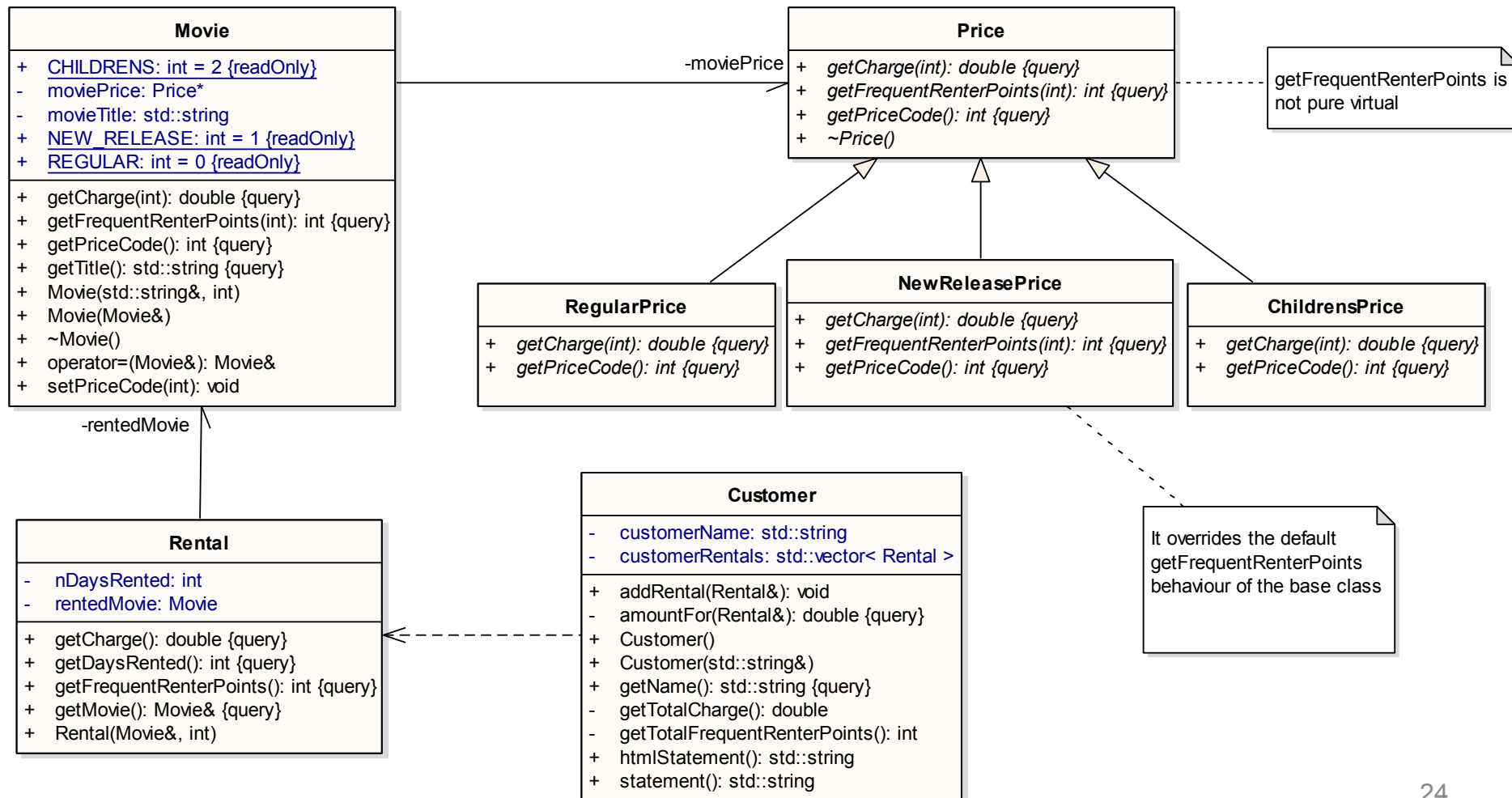
When done with all the legs, make *Price::getCharge* a pure virtual function

# Step 15

Apply the same procedure to *getFrequentRenterPoints*

In this case do not make the superclass method pure virtual  
Create an overriding method for the new releases and leave  
a defined method (as the default) on the superclass

Step 15 - End of refactoring



# Step 15: The End

Step 15 - Sequence diagram at the end of refactoring

