McGill

Refactoring

Chen Tang

March 3, 2004

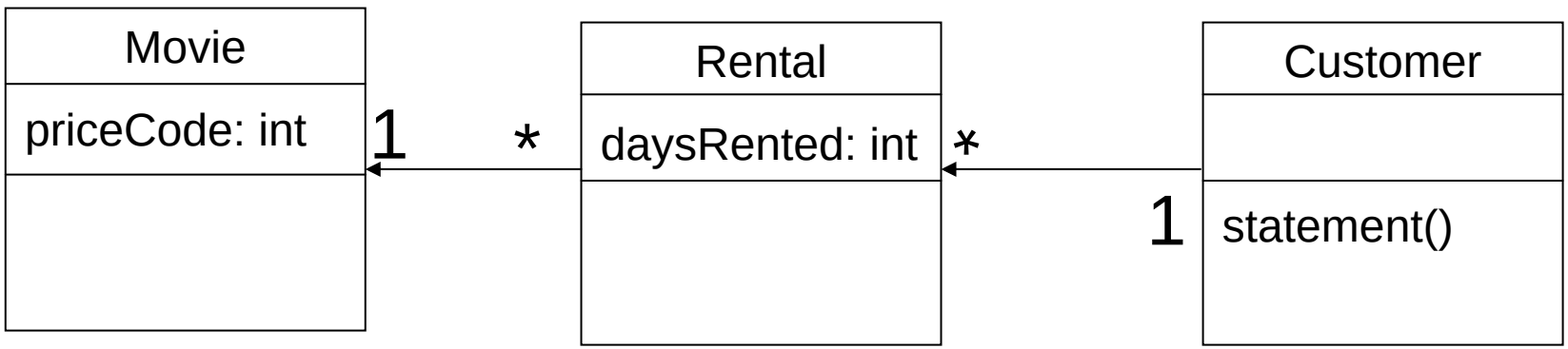
What Is Refactoring (Definition)

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

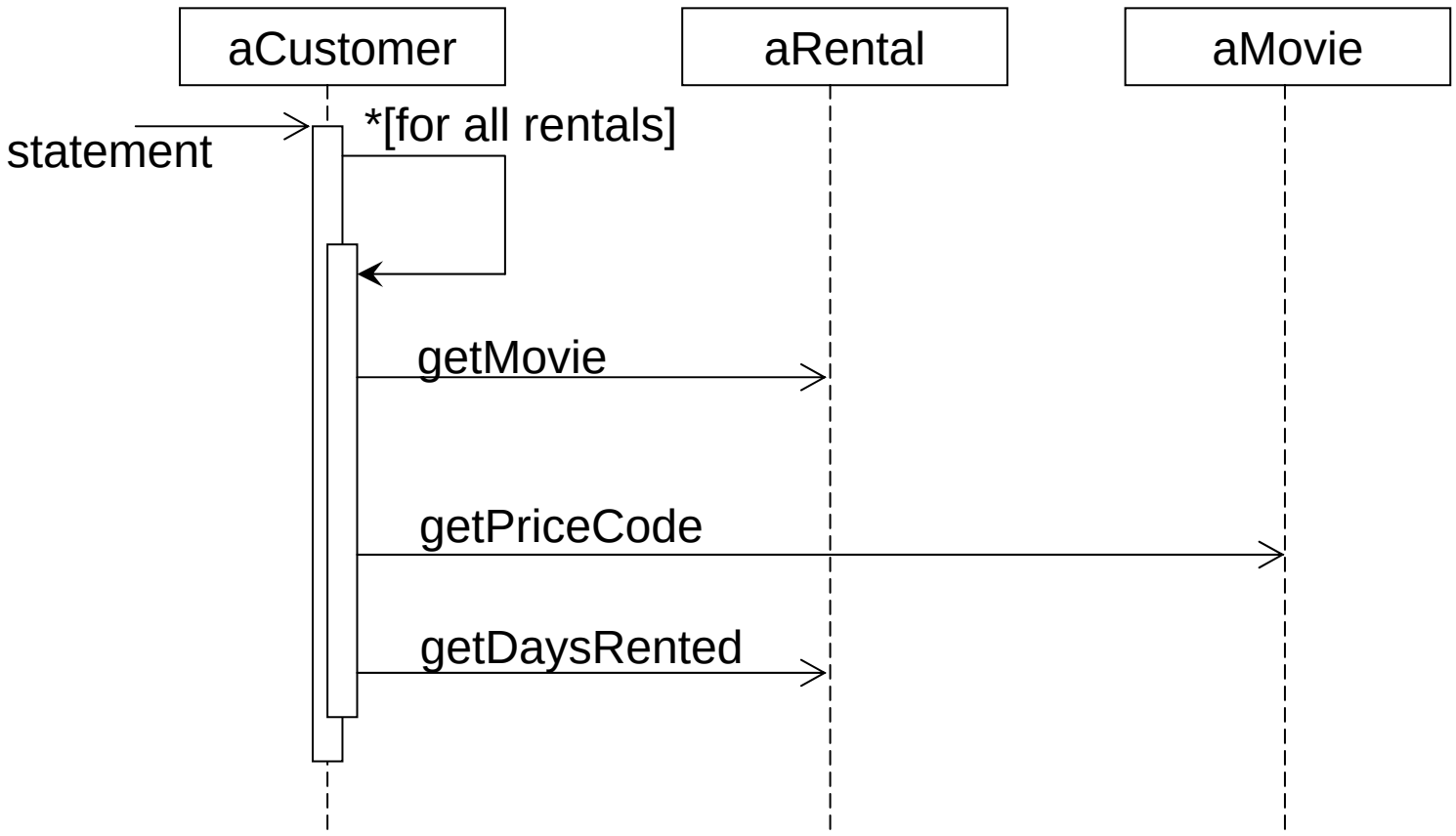
Where Did Refactoring come From

- Good programmer certainly have spent some time cleaning up their code.
- Two of the first people recognize the importance of refactoring: Ward Cunningham and Kent Beck
- Bill Opdyke put the application of refactoring to even broader area.
- John Brant and Don Roberts have taken the tool ideas in refactoring much further to build the Refactoring Browser

A Practical Example of Refactoring (1)



A Practical Example of Refactoring (5)



A Practical Example of Refactoring (6)

Problems:

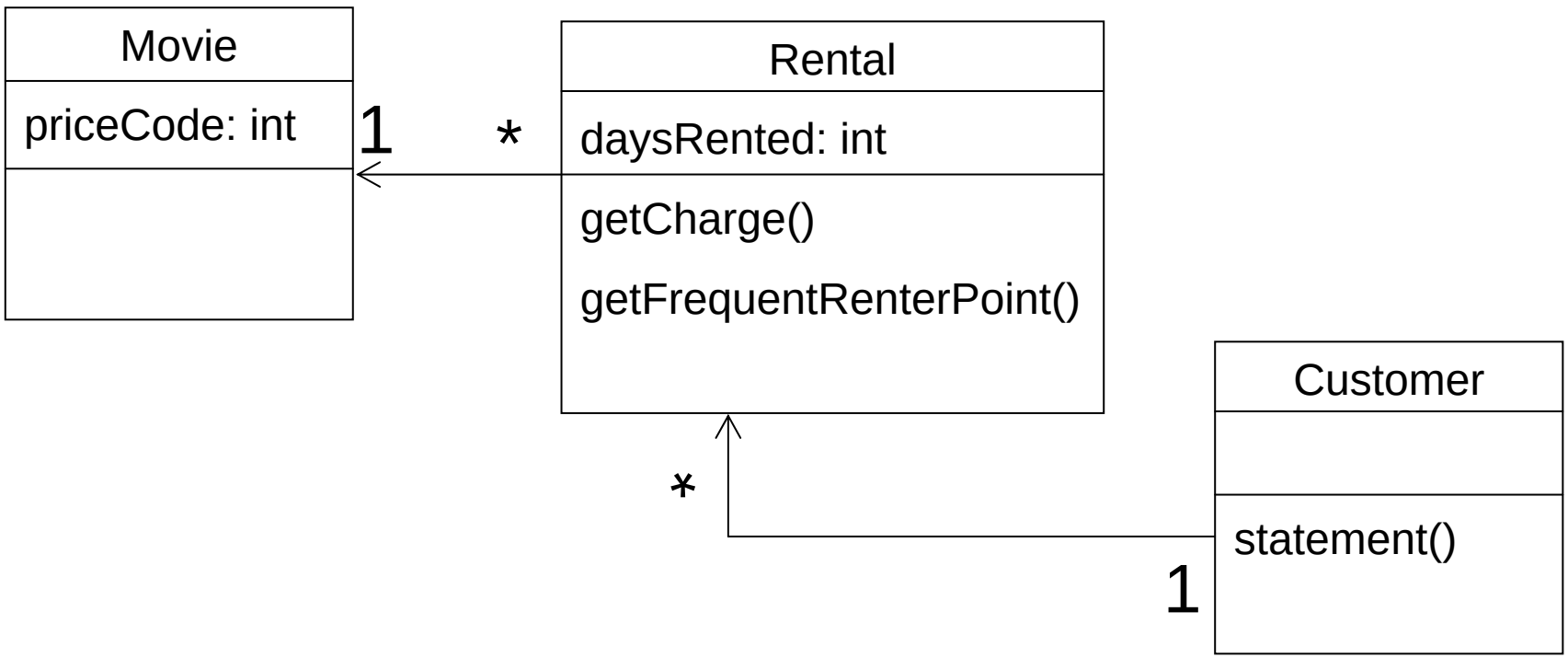
- First, there too few responsibilities put into the class Movie and class Rental. And statement method in class Customer does way too much!
- Second, hard to add new functions. Say want to generate bill in the form of HTML!
- Third, calculate every time's renter point should be the class Rental's responsibility!

A Practical Example of Refactoring (7)

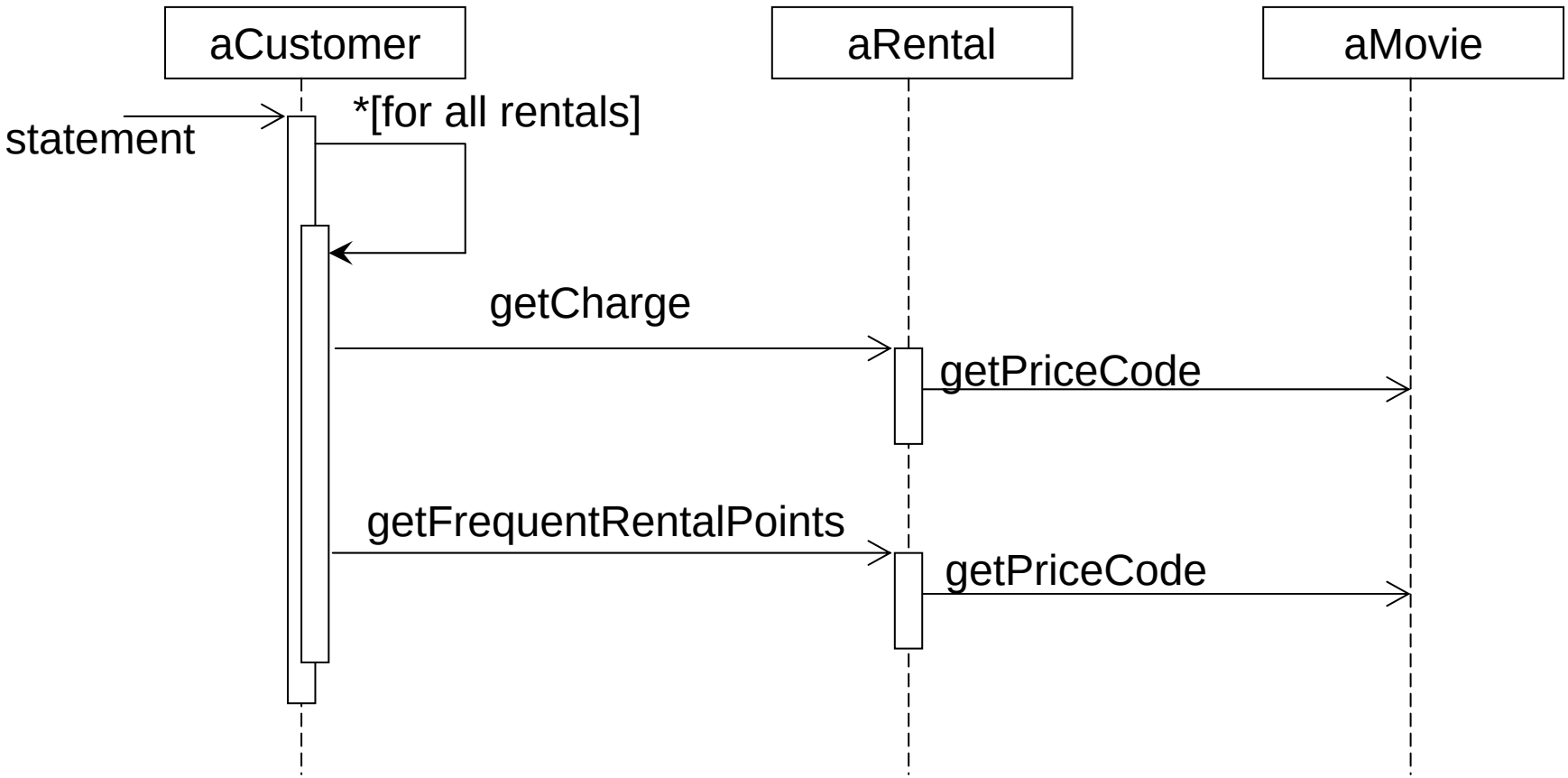
Refactoring (*Moving Method*):

- Moving the Price calculation for every rental to class Rental
- Moving the calculating renter points to class Rental

A Practical Example of Refactoring (7)



A Practical Example of Refactoring (8)

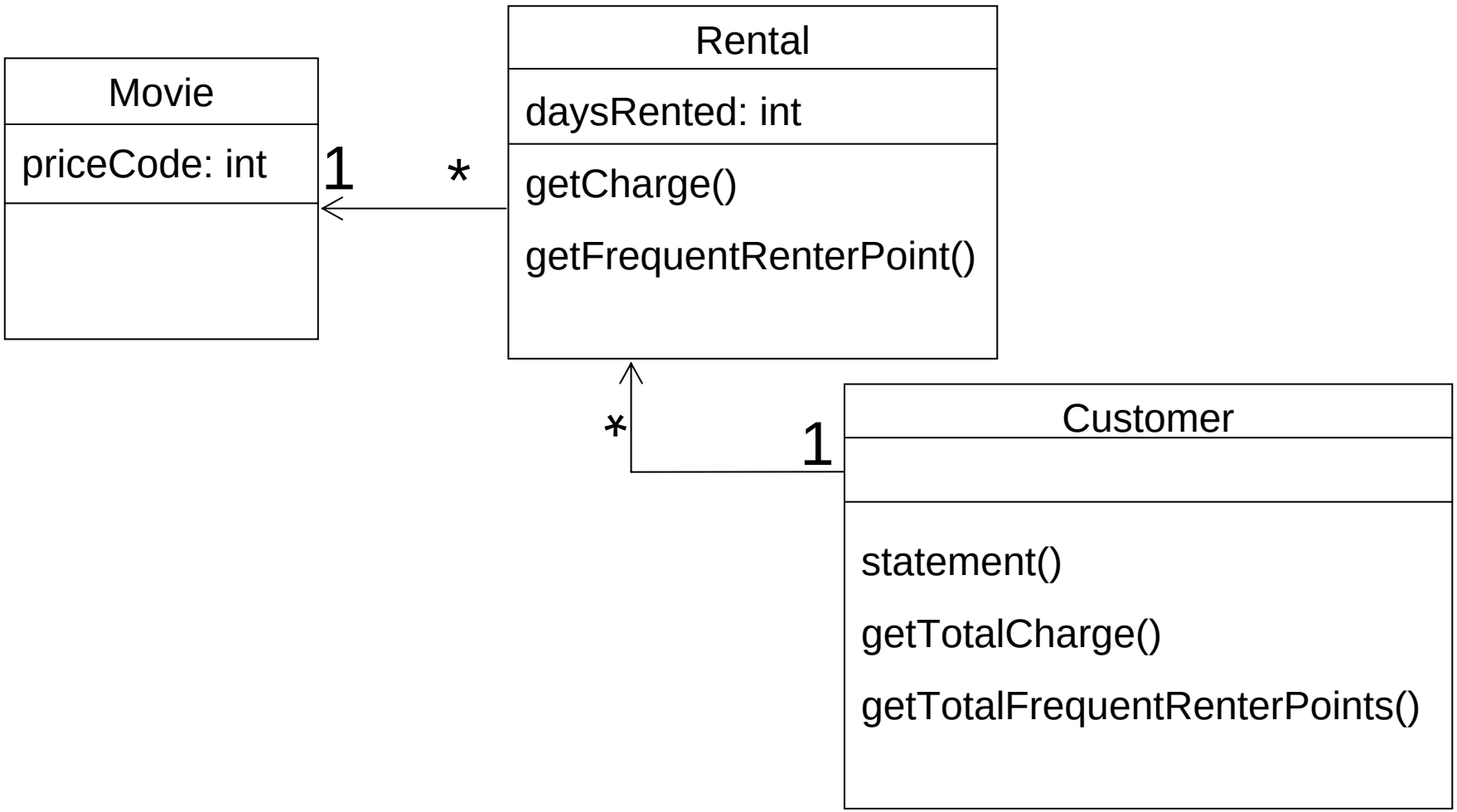


A Practical Example of Refactoring (9)

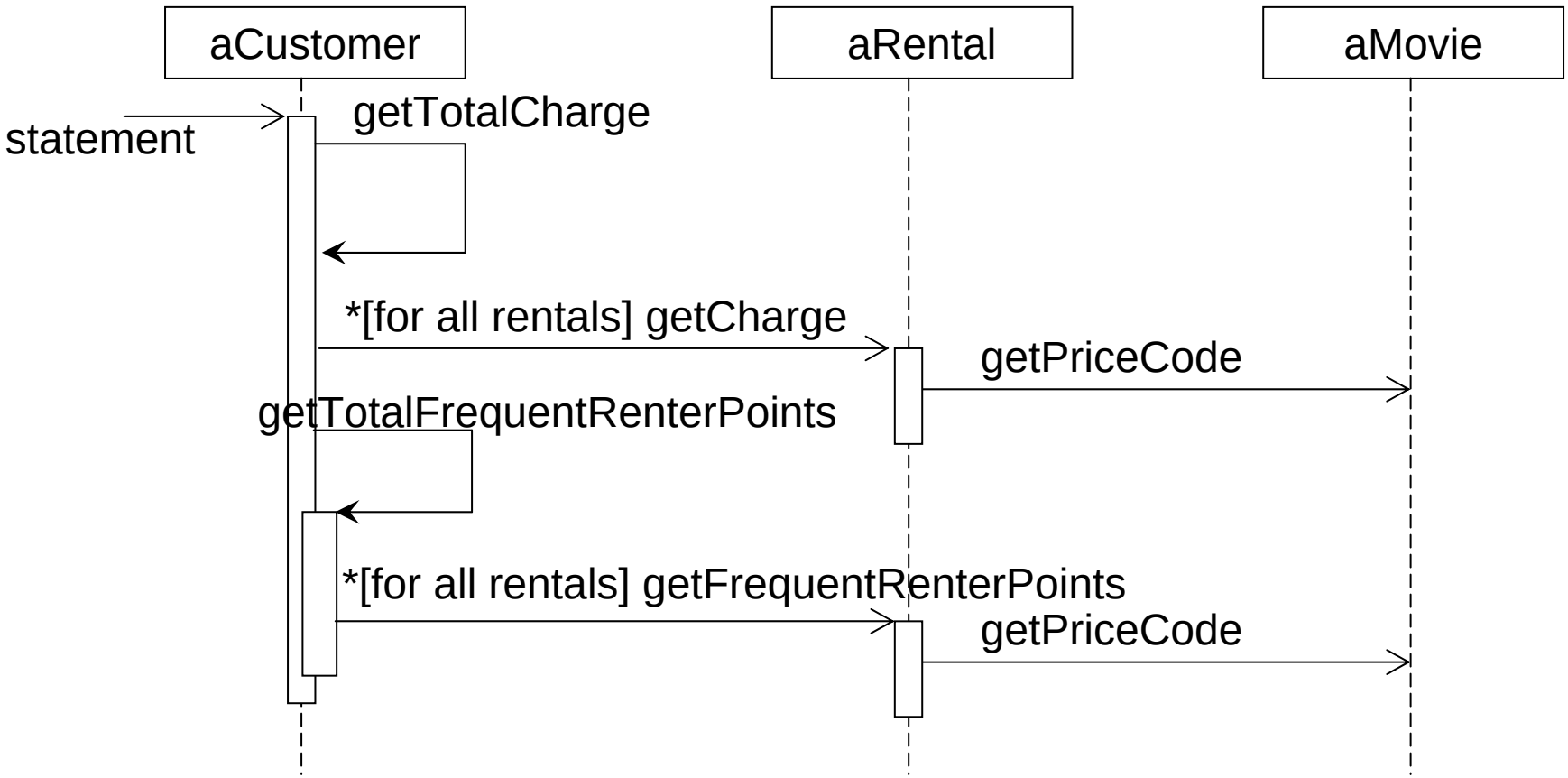
- Removing Temps. There are two temporary variables in statement method: totalCharge & totalRentalPoints.

  Replace Temp with Query:
  getTotalCharge()
  getTotalFrequentRenterPoints()

A Practical Example of Refactoring (10)



A Practical Example of Refactoring (11)



A Practical Example of Refactoring (12)

Now it’s very easy to add another method to generate statement in HTML format:

```
htmlStatement() {
    //call getTotalCharge to get total charge
    //call getFrequentRenterPoints to…
}
```

Why Should We Refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

The Importance of Building Tests

Whenever we do refactoring, the first step is always a solid test for that section of code:

- We have to rely on tests to tell us whether we introduce any bugs and whether the behavior of the software changes!

When Should We Refactor?

- The Rule of Three
  - Third time you do something similar you refactor.
- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Bad Smells in Code

1. Duplicate Code
2. Long Method
3. Large Class
4. Long Parameter List
5. Divergent Change
6. Shotgun Surgery
7. Feature Envy
8. Data Clumps
9. Primitive Obsession
10. Parallel Inheritance Hierarchies

Bad Smells in Code (cont'd)

11. Lazy Class
12. Speculative Generality
13. Temporary Field
14. Message Chains
15. Middle Man
16. Inappropriate Intimacy
17. Alternative Classes with Different Interfaces
18. Incomplete Library Class
19. Data Class
20. Refused Bequest
21. Comments

Some Refactoring Rules

Composing methods

- Extract Methods
- Inline Methods
- Inline Temp
- Replace Temp with Query
- Introducing Explaining Variable
- Split Temporary Variable

Moving Features Between Objects

- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man

Organizing Data

- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Encapsulate Field

Dealing with Generalization

- Pull Up Field
- Pull Up Method
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy

The “Two Hats”

Kent Beck’s metaphor of two hats:

When using refactoring to develop software, we divide our time between two activies:

- Adding new functions
- Refactoring

Problems with Refactoring & When shouldn't do refactoring

- Databases
- Changing Interfaces
- Design Changes That Are Difficult to Refactor
- When Shouldn't do refactoring?
  - When you should rewrite from scratch instead
  - When you are close to deadline

Refactoring Tools

One of the largest barrier to refactoring is woeful lack of tool to support it.

- Tool can make refactoring less a separate activity from programming.
- Tool can make design mistakes less costly, because it makes refactoring easy and less expensive.
- Much less testing will be needed as some refactoring can be done automatically, though not all of them. So tests are still indispensable!

Towards Next Step (Tentative)

Integrate the following refactoring rules into AToM3 by means of graph rewriting (another alternative will be hard coded):

- Pull Up/Push Down field
- Pull Up/Push Down Method
- Extract Subclass/Superclass/Interface
- Rename method
- Hide Method
- Change direction of Association Between Class

Further References

- Martin Fowler, Refactoring improving the design of existing code, Addison Wesley, 1999
- Refactoring Home Page: http://www.refactoring.com/
- refactoring catalog: http://www.refactoring.com/catalog/index.html

Thanks!

Questions & Comments ?