Lecture 26: Refactoring (part 2)

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 6448 - Spring Semester, 2002

Credit where Credit is Due

- Some of the material for this lecture and lecture 25 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 more complete tutorial on refactoring using a slightly larger example that requires multiple refactorings
- Quiz
  - A design phase for a system has finished and it is now time to start implementation; your manager decides that use cases 5, 15, and 35 are to be implemented during the first iteration; assuming that all of the different types of models discussed in chapters 2, 4, and 6 of the textbook have been created for the system, how should the developers use this information to start their implementation efforts?

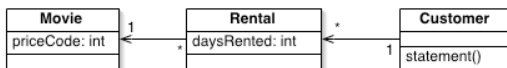
Tutorial

- 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 page 4 of Refactoring

Step 1: refactor statement()

- Why? It's a "long method" which is one of the "bad smells" presented in the last lecture
- Besides, our purpose is to add a new method to generate an HTML statement and refactoring statement() may lead to code that can be reused by the 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
  - which are easier to manage and move around
- 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 to the new method (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

© Kenneth M. Anderson, 2002

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!

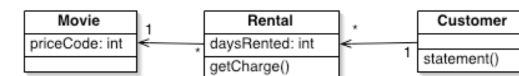
© Kenneth M. Anderson, 2002

Step 3: move method

- amountFor() uses information from the Rental class, but it does not use information from the customer class where it is currently located
  - methods should be located close to the data they operate, so lets move amountFor() to the Rental class
    - we can get rid of the 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!

© Kenneth M. Anderson, 2002

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!

© Kenneth M. Anderson, 2002

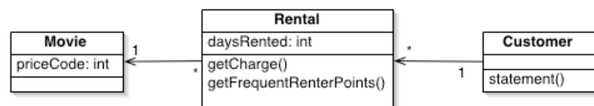
Step 4: Replace Temp with Query

- 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 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 assingment 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



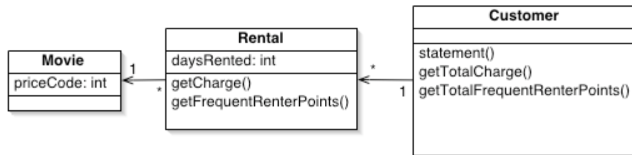
Customer continues to get smaller, Rental continues to get larger; but Rental now has operations that change it from being a "data holder" to a useful object

Our sequence diagram has changed (see page 25); 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 tons of code whose purpose may not be clear

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 with the existing statement() method and the planned htmlStatement() method

Our sequence has again changed, see page 31, 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!

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

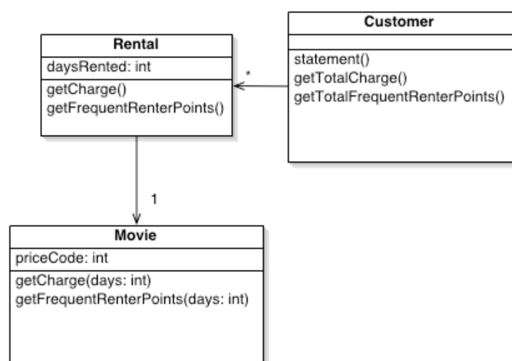
New Requirements

- 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 a parameter
- Step 8b: move getFrequentRenterPoints() to Movie
  - ditto!

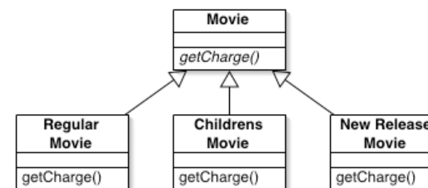
Current class diagram



Movie has new methods, finally making the transition from data holder to object;

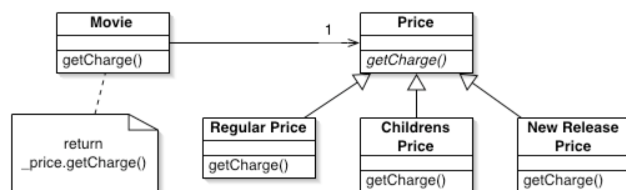
these methods will allow us to handle new types of movie easily

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 childrens movie. So this approach won’t work!

The 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, and introducing a bug, so you can be sure that the subclass’s method is the one being invoked!
- After you have done the move, change Price’s getCharge to an abstract method
- Don’t forget to test; everything still works!

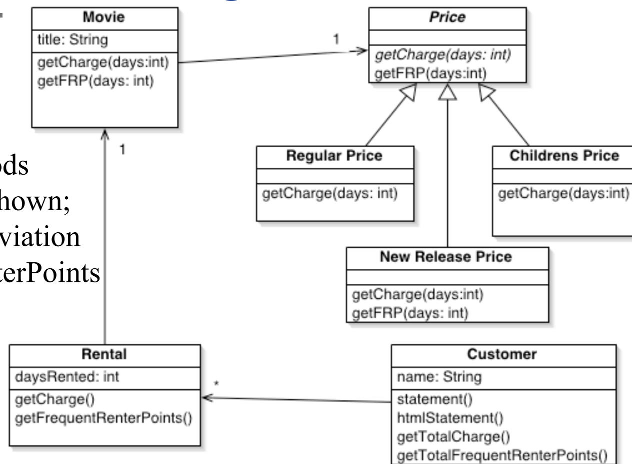
Step 12: handle renter points

- 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 page 51 for final sequence diagram

What's Next?

- Now that we have covered design patterns and refactoring, we will next look at some Object-Oriented Design Heuristics
  - Rules of thumb that help us distinguish between "bad" OO designs and "good" OO designs