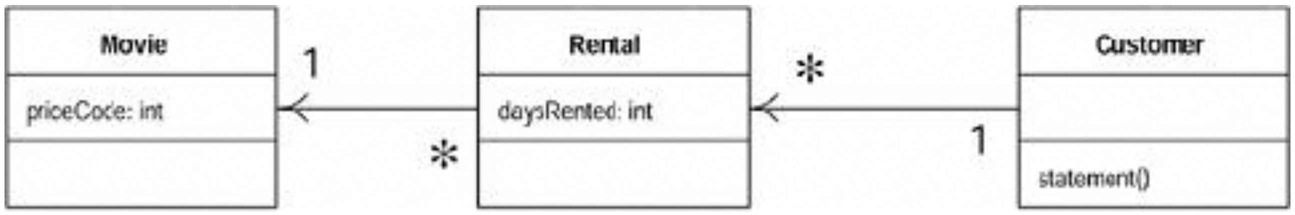
Refactoring exercise - Step by step

1. Start Eclipse and import the example code
2. Look at the production code and test code. This is the structure



3. Look at the Customer.statement() which contain Business Logic, presentation and classification logic. Clearly it needs to be cleaned. It has too many responsibilities.
4. **Look at the tests** in CustomerTest to see functionality
5. **Run tests** and see them work.
6. **Examine the system**.
   We have a Customer class with a list of Rentals.
   In Rental we keep track of number of days rented and reference to the movie rented.
   In Movie we keep track of title and priceCode.
   Both Rental and Movie looks like **simple data structures** with no behaviour
   All behaviour is in Customer.statement().
7. The statement() method loops over each rentals,
   for each rental it:
   - calculates the amount (or cost)
   - calculates frequent renter points, based on BL and classification system
   - produces an output for the rental
   Eventually produces a footer line with total amount and the earned rental points
8. This is a pretty **tangled implementation**!
   We are supposed to add an html_statement() method output to the application. Just copying the statement() and modify it will give **lots of duplications**. A better approach is to see if we can **remove/move some of the responsibilities** of the statement() method.
9. One good candidate, is to **extract the calculation** of the amount for each rental.

```
// Determine amounts for each line
double thisAmount = 0;
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2) {
            thisAmount += (each.getDaysRented() - 2) * 1.5;
        }
        break;
    case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented() * 3;
        break;
    case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3) {
            thisAmount += (each.getDaysRented() - 3) * 1.5;
        }
```

```
            break;
    }
```

10. This seems to be a well defined responsibility. Extract it into a specific method, a.k.a "Extract Method Refactor". Copy the code block and create a new method
    ```
        private double amountFor(Rental each);
    ```
    Remove the comment since the method name communicates the purpose.
    Replace the original code block with a call to the extracted method.
    Run tests !
11. This "Extract Method Refactor" could have been performed with **eclipse tool support**.
    Undo a couple of steps and perform with eclipse tool support.
    Select the code block, left click and chose Refactor>Extract Method…
    Name the method "amountFor"
    Remove the comment and **Run tests !**
12. Look closer at the extracted method amountFor(), some of the naming doesn't make sense. **Rename local variable** thisAmount to result. Rename the "each"-parameter to rental.
    Run tests !
    Is it worth renaming variables? Clearly it is, it can increase the readability substantially and makes the communicate its purpose clearly.
13. Now, the `amountFor()` method, it is concerned with a rental (**Feature envy**) and should be moved to Rental.
    This step could be done by ordinary "Cut, Copy and Paste" between the Customer and Rental classes. Alternatively, apply the "Move Method…"-refactor to move `amountFor()` from Customer class to Rental class, make it public, remove the parameters and rename it to `getCharge()`. If the later is used you need to skip step 14-16, but of course **Run tests !**
14. Start by copying the `amountFor()` from Customer and paste it in Rental, make it public and rename it to `getCharge()` fix compilation issues in Rental (get rid of the rental parameter).
15. Change remaining implementation of amountFor() in Customer to just delegate by returning rental.getCharge();
    Run tests !
16. Then remove the amountFor() method since it is only called once from statement() and replace the call amountFor() with each.getCharge()
    Run tests !
17. In statement() the temp variable "thisAmount" is redundant, only used twice at same line. Replace with the query each.getCharge()
    Run tests !
    Getting rid of temp variables are good because it clutters the code, of course it could be a performance penalty here because we do the query twice… but this could be optimised in the rental class…
18. // **Don't do this step!**… it is part of another alternative exercise.
    // ~~In the loop in the statement() method, rename "each"-loop variable to rental~~.
19. Let's see if we can factor out some more responsibilities from the statement() method, frequentRenterPoints is a good candidate.
    Factor out method returning number of points collected as
    frequentRenterPoints += getFrequentRenterPoints(each);
20. The implementation of getFrequentRenterPoints(…) could be simplified, **change implementation** to return either 2 or 1 points to make it more readable.

21. Looking at the getFrequentRenterPoints(Rental each) we can see that it is more concerned about the Rental than the customer (**Feature envy**), hence move it to Rental class.
22. **Copy the method and move it**, then change original implementation to delegate and return each.getFrequentRenterPoints();
    Run tests !
23. **Eliminate** the getFrequentRenterPoints(Rental each) in the Customer class by directly calling the Rental each.getFrequentRenterPoints();
24. In statement we still got two **temporary variables**: totalAmount and frequentRenterPoints. Let's get rid of them by replacing them with queries. Start with totalAmount.
25. **Replace** the totalAmount with a private double getTotalCharge() method where we duplicate the for-loop and calculates the total amount. Now we can remove the totalAmount variable in statement() and replace the usage with a query getTotalCharge().
    Rerun the tests !
26. Do the same with frequentRenterPoints. **Replace** it with private int getTotalFrequentRenterPoints(). Replace all frequentRenterPoints with a getTotalFrequentRenterPoints() query.
    Run the tests !
27. Now we are in a shape where statement() **only contains presentation logic**. Now we can **switch hat** from "refactoring" to "feature adding" hat and start implement the html implementation.
28. As always!! We **start by implementing a test** for the new feature. Go to CustomerTest class. Let's add a test for the to be html statement method. Do it by copying testStatementForCustomer1(). Rename the copy to testHtmlStatementForCustomer1() and call the to be implemented htmlStatement() method instead. **Cheat** a bit by copy in the expected string with new html strings and implement the htmlStatement() as a copy of statement().
    Rerun the tests to see that there is a failure !
    The cheat code needed could be found in the ReadMe.md file in the Eclipse Project.
29. Cheat by pasting in an **implementation** and rerun the tests again. Fix accordingly. The cheat code needed could be found in the ReadMe.md file in the Eclipse Project.
    ———— First feature done —————

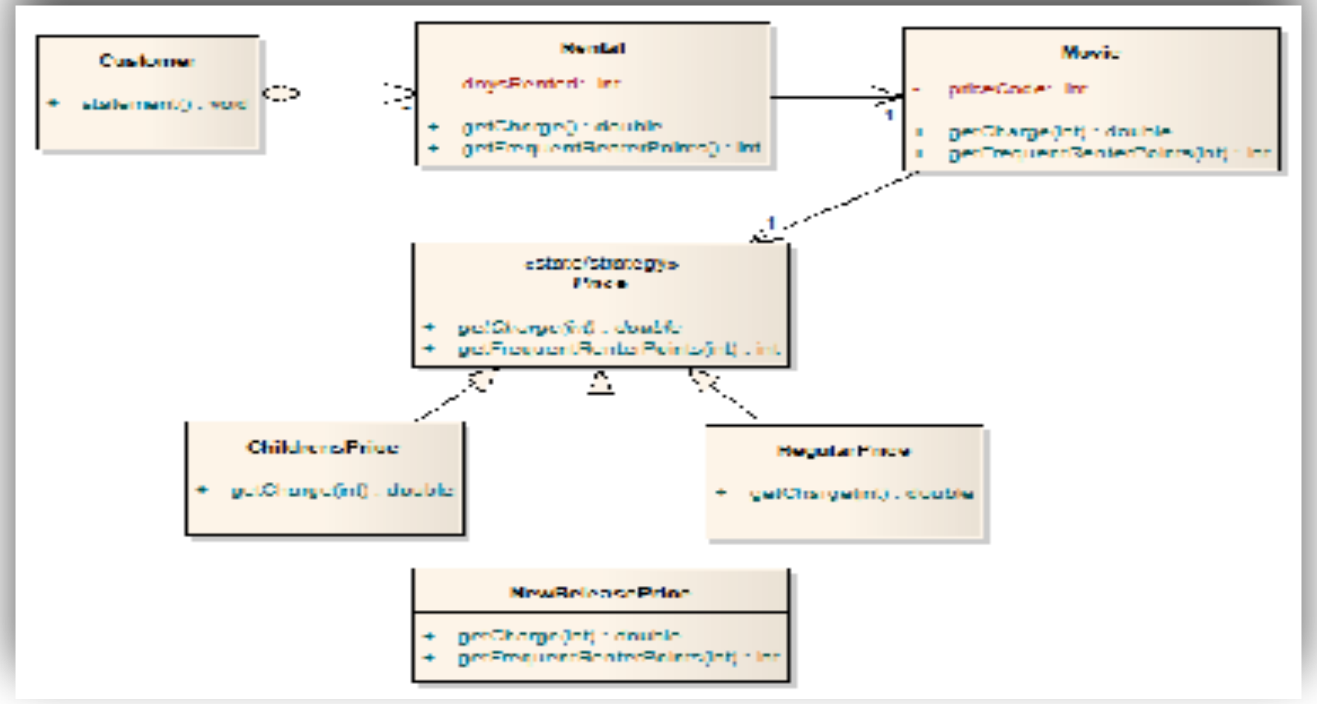
———— Second feature ————— (Altering the Movie classification structure)

1. First, **switch to refactoring hat**.
2. In Rental class we switch and evaluate on foreign data that belong to the movie class. This is vulnerable and switch:es are ugly and if you need to use them, **make sure you own the data you switch on**. I.e the data is in the same class…
3. Change the implementation of getCharge() and getFrequentRenterPoints() to delegate the switch and if conditions to the movie class instead. (Do a "Move-method" refactoring). Observe that the getDaysRented() needs to be parametrised to the getCharge method.
4. Looking at the Movie class, we could see that it uses numeric price code, and conditional logic to evaluate via a switch statement. This makes the business logic **highly coupled** to the classification system. Meaning if the classification system changes (new constants being added, means that the BL has to change.
5. We would like to **replace the switch statement** with some inheritance mechanism. One way to do that would be by subclassing the movie with subclasses for the different movie categories and use polymorphism. This will not work good since the classification can change over the lifetime of a movie (New release -> Regular).
   A better approach is to use the State (or **Strategy**) pattern and inheritance as a strategy pattern. That pattern will use a separate object to calculate the price.
   Let's introduce a Strategy pattern by first isolating the priceCode and replacing it with a Strategy. We have a setter and a getter for priceCode. In the constructor use the setter for the priceCode. **Run the test !**
6. Now we are ready to add a Strategy. Add a new abstract base class in the production code. Name it Price, make sure to check the abstract checkbox
7. Add a behaviour for getting the priceCode:
   public abstract int getPriceCode();
8. Add concrete classes for RegularPrice extending Price.
9. Override the getPriceCode() to return the Movie.REGULAR constant.
10. Do the same for ChildrenPrice and NewReleasePrice classes. Don't forget to extend Price and refactor as needed.
11. Now back to our Movie class. Replace the field int priceCode with a strategy:
    private Price price;
12. Modify the getter to delegate to the strategy by doing: return price.getPriceCode();
13. In the setter we need to use a switch statement (on newPriceCode) to be able to create the proper instance. Now we have switched the priceCode implementation to a Strategy pattern. **Run the tests !**
14. The next step is to move the Movie.getCharge() calculation into the Strategy (into the abstract Price class). Copy the method into the abstract class Price.
15. Replace the implementation in Movie with a delegation:
    return price.getCharge(daysRented);
    Run the tests !
16. In the the abstract Price we now can replace the switch statement with polymorphism. We do this by pushing down the implementation of getCharge() to the subclasses. Let's do it, one leg at a time…
    Copy the implementation from the abstract Price to RegularPrice and add the @override annotation
    Rip out the switch statement leaving only the body for the RegularPrice. Simplify if needed.
    Do the same for the other legs.
17. Once we push the implementation down to each leg, we can get rid of the implementation in the abstract class.
    Run the tests !

18. Do the same with the frequentRenterPoint calculation. Copy it and move it to abstract Price class and replace the implementation in Movie with a delegation:
    return price.getFrequentRenterPoints(daysRented);
    Run the tests !
19. Now looking at Price, we can see that getFrequentRenterPoints() only has a specialisation in the case of NEW_RELEASE. In all other cases it has the same implementation (return 1;) Push down implementation only to NewReleasePrice, DON'T forget the @override annotation.
20. Simplify the implementation in subclass and abstract class.
    Run the tests !

What we have seen in this exercise is that a sequence of transformations could lead to a quite substantial change of the internal structure. And it has all been done in a safe way.

Was it worth it? I definitely think so. The code is **easier to modify**, since the dependencies has been cut off and the code is **easier to read**.
Final state of the object model



Customer
+ statement() : void
Rental
- daysRented : int
+ getCharge() : double
+ getFrequentRenterPoints() : int
1
Movie
- priceCode : int
+ getCharge(int) : double
+ getFrequentRenterPoints(int) : int
1
«abstract/strategy» Price
+ getCharge(int) : double
+ getFrequentRenterPoints(int) : int
ChildrensPrice
+ getCharge(int) : double
RegularPrice
+ getCharge(int) : double
NewReleasePrice
+ getCharge(int) : double
+ getFrequentRenterPoints(int) : int