3F6 - Software Engineering and Design

# Handout 8

# Design Patterns (III) and Refactoring

With Markup

Ed Rosten

## Contents



Copies of these notes plus additional materials relating to this course can be found at:
`http://mi.eng.cam.ac.uk/~er258/teaching`

# Visitor Pattern

## Problem

Sometimes the set of classes in a hierarchy has become well defined and fixed, but the set of operations defined by the hierarchy is still fluid.

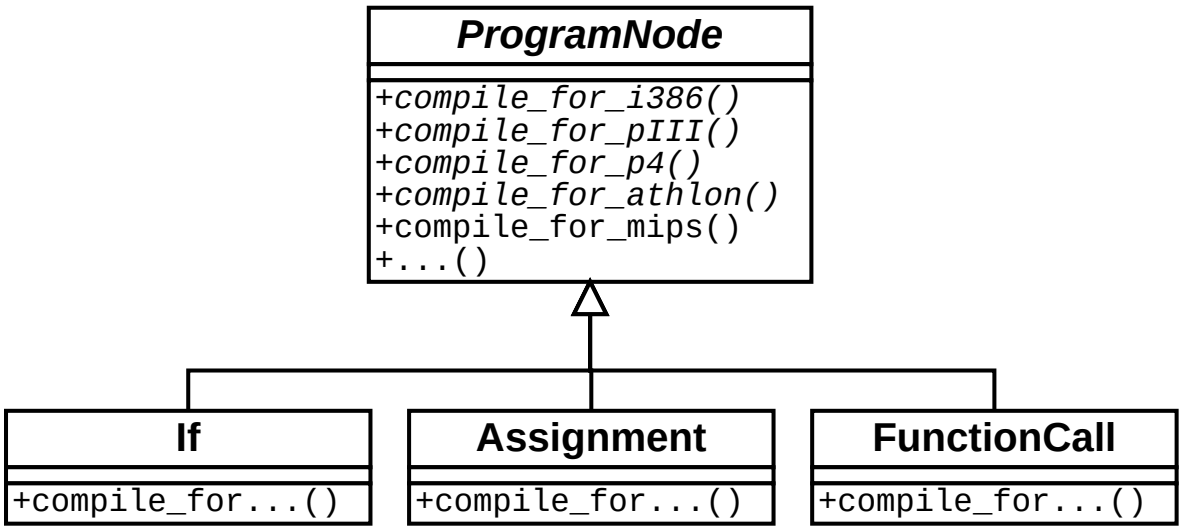
For example we may have a simple text document composed hierarchically of paragraphs, lines, words and characters (using the composite design pattern) which we wish to prepare for printing on any one of a large number of printing devices, or we may wish to convert it into any one of a large number of word processing formats.

Alternatively, we may have a class hierarchy representing the various kinds of nodes present in the parse tree for computer programs in some language. This representation of a computer program could be used for investigating various methods of optimising the code or compiling it for different architectures (cf the expression evaluator in lecture 5).

In both of these contexts, the class hierarchy is well defined but the interface that it must present is likely to change.

## Solution 1

Simply add new functions as they are needed into each class in the hierarchy.



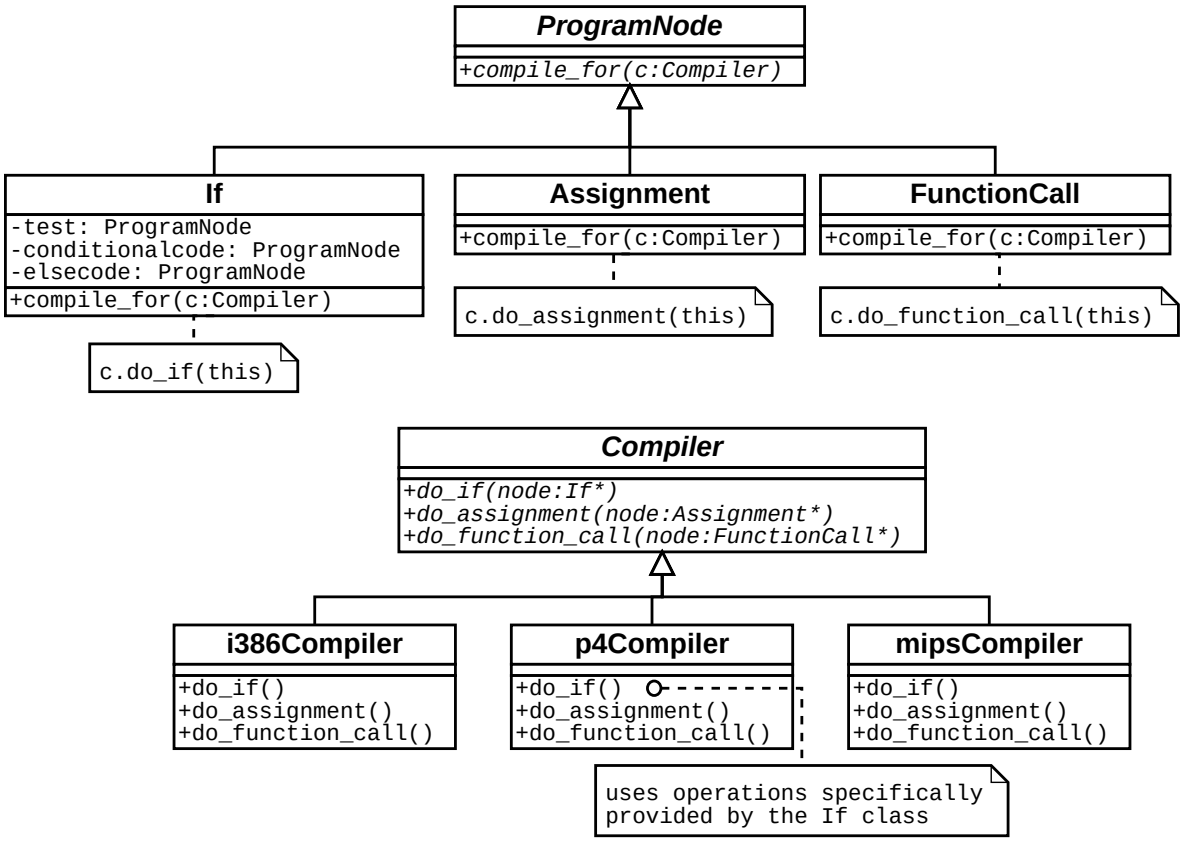

| | | |
|---|---|---|
| Pros: | - | simple for very simple systems |
| Cons: | - | a significant number of edits will be required to add a new function |
| | - | the code that represents the new operation is distributed between several different classes. |

It would be preferable if all the code for the new function could be in a single place. This would make it easier to implement, test, and maintain.

## Good Solution

Place a single function into each of the classes in the hierarchy which allow them to be processed by an external object. The latter contains all the code for processing objects belonging to the class hierarchy.

This external object has to belong to a class from a hierarchy which contains functions that know how to handle each class in the original hierarchy. This structure allows us to create new classes in the new hierarchy which can process objects from the original hierarchy in a new way without further changing any of the code in the original class hierarchy.



## Design Pattern

This is an example of the **Visitor** pattern and is interesting for two reasons. First, it provides an example of a class hierarchy in which the subclasses conceptually correspond to actions rather than kinds of objects present in the problem domain. Secondly, it shows how polymorphic behaviour can be extracted from a class hierarchy and implemented externally in an extendible and flexible way.

This is called the Visitor pattern because an object from the `Compiler` class hierarchy *visits* objects from the `ProgramNode` hierarchy. The `ProgramNode`s don't know anything specific about their visitor, only that it is a kind of `Compiler`.

Thus this pattern allows us to generate a table of functions which are selected at run time based on *both* the type of `ProgramNode` *and* the type of `Compiler`.

# Refactoring

Refactoring is the process of redesigning code so that its behaviour is unchanged but [0]

- its structure is improved
- it is easier to read and understand
- overall code size is (usually) reduced

The key idea is to make a series of (provably correct) small changes to the code, with continuous testing to detect errors as soon as they occur. Note that the availability of comprehensive regression tests is an essential prerequisite for refactoring.

The benefits of refactoring include:

- More efficient detection of bugs.
- Simpler addition of new functionality.[1]
- Easier and more effective maintenance.[2]

Refactoring is often repeated several times in the lifetime of a software project.

0. Refactoring is in many ways a formalisation of normal sofware refinement.
1. The need for a new feature is a common motivation for refactoring.
There may be nothing wrong with the existing code until a new requirement arises.
2. Refactoring after the addition of new code will typically lower the total
cost of future maintenance.

# Refactoring Tools

The original methodology of refactoring was originated in William Opdyke's PhD thesis and expounded further in the book

**Refactoring: Improving the Design of Existing Code**, Martin Fowler, Booch, Jacobson and Rumbaugh, 1999

This book describes a set of manual approaches to refactoring.

There are two main reasons for refactoring

- cleaning up bad code - usually irreversible.
- changing design decisions - usually reversible since several design options might be reasonable.

For cleaning up bad code, a variety of tools have been developed (see `http://www.refactoring.com` )

These typically work by storing a library of patterns (*bad smells*) and associated translations which will improve the code.

# Bad Smells

### Duplicated Code

Two or more chunks of code that are almost the same is a strong indicator that a single function should be used from both places.

### Long Functions

Break long functions into shorter more manageable chunks. Examples of duplicated code often occur in long functions.

### Long Parameter List

Functions with more than 4 or so parameters should be avoided. Bundle related parameters into structs or split the function into several simpler functions.

### Feature Envy

If a member function (operation) is more interested in the contents of another class than its own, move it to the other class.

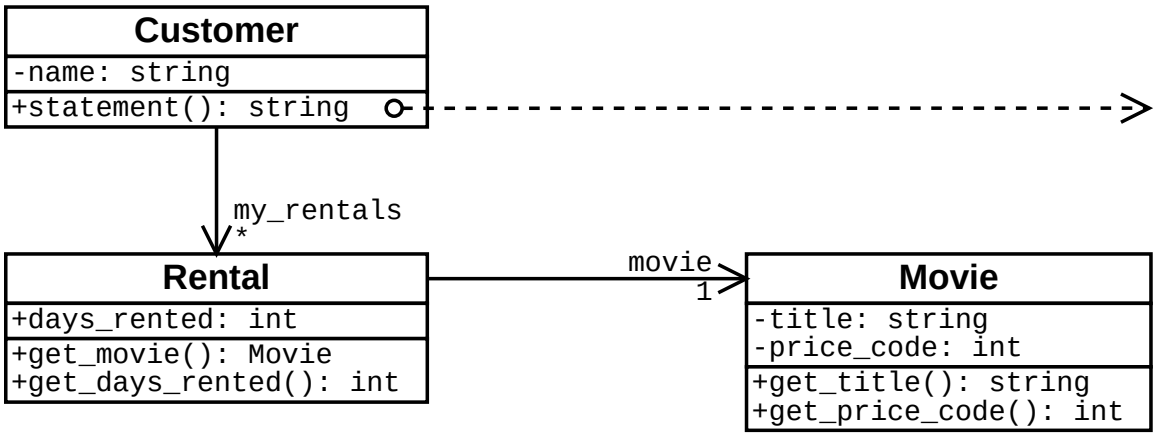
### Switch Statements or Type Codes

These are often indicators that a class is really trying to represent several discrete concepts and is therefore a candidate for using polymorphism to achieve the varying behaviour.

The Fowler book lists many more bad smells.

# Worked Example

Consider some software for managing a movie rental business. A UML class diagram is given below.




The method `Customer::statement` is very long indicating a bad smell and a strong candidate for refactoring.

The first step is to extract part of the function and make it a separate function.

Note that this example can also be seen as motivating good design decisions (by presenting a bad design first and then modifying it so that the design is cleaner and better).

```
string Customer::statement(){
  float total_amount=0;
  int frequent_renter_points=0;
  list<Rental*>::iterator rentals;
  string result = "Rental record for " + name;

  // go through all the rentals
  for(rentals = my_rentals.begin(); rentals!=my_rentals.end();rentals++){
    Rental* each = *rentals;

    // determine the amount for each rental
    float this_amount=0;
    switch(each->get_movie()->get_price_code()){
      case Movie::REGULAR:
        this_amount += 2;
        if(each->get_days_rented() > 2){
          this_amount += (each->get_days_rented() - 2) * 1.5;
        }
        break;
      case Movie::NEW_RELEASE
        this_amount += each->get_days_rented() * 3;
        break;
      case Movie::CHILDRENS
        this amount += 1.5;
        if(each->get_days_rented() > 3){
          this_amount+=(each->get_days_rented()-3) * 1.5;
        }
        break;
    }
    total_amount += this_amount;

    // add frequent renter points
    frequent_renter_points++;
    // add a bonus for two day new release rental
    if((each->get_movie()->get_price_code() == Movie::NEW_RELEASE)
       && (each->get_days_rented()>1)) {
       frequent_renter_points++;
    }

    // add figures for this rental to the statement
    result += each->get_movie()->get_title();
    result += "  " + this_amount + endl;
  }

  // add footer lines to statement
  result += "Amount owed is " + total_amount + endl;
  result += "You earned " + frequent_renter_points + " frequent renter points"

  return result;
```

# Extracting a function

This is an example of the "Extract Method" of refactoring. To illustrate the methodical, step by step nature of the approach, we will look at this in detail. The mechanics of applying the Extract Method are:

1. Identify a section of code for extracting. We will extract the code for determining the price for each rental.
2. Create a new member function and give it a good name. We will call the function `amount_for()`.
3. Copy the section of code into the new function.
4. Find all the variables used in the section of code. In this example, there are two: `this_amount` and `each`. `each` is not modified in the code section, so it can be passed in to the new function as an argument. `this_amount` is assigned to within the code segment, so we can make this the return value of the function.
5. Replace the original code segment with a call to the new function
6. Compile and test

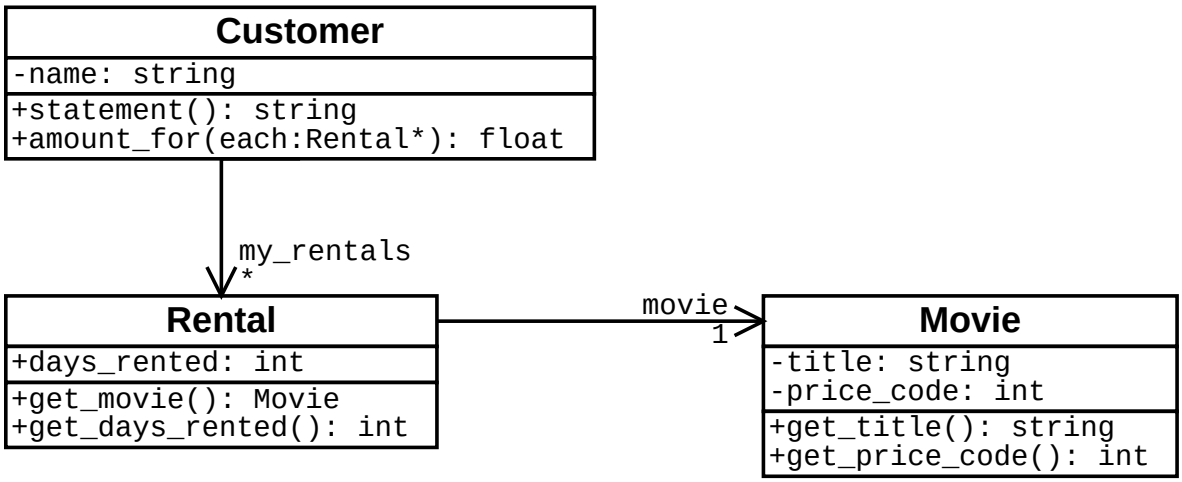
The new function now looks like:

```
float Customer::amount_for(Rental* each){
  float this_amount=0;
  switch(each->get_movie()->get_price_code()){
    case Movie::REGULAR:
      this_amount += 2;
      if(each->get_days_rented() > 2){
        this_amount += (each->get_days_rented() - 2) * 1.5;
      }
      break;
    case Movie::NEW_RELEASE
      this_amount += each->get_days_rented() * 3;
      break;
    case Movie::CHILDRENS
      this amount += 1.5;
      if(each->get_days_rented() > 3){
        this_amount+=(each->get_days_rented()-3) * 1.5;
      }
      break;
  }
  return this_amount;
}
```

and the original code section is replaced by

```
float this_amount = amount_for(each);
```

As a class diagram we now have:



# Moving a function

If we examine the new function further, we find that it does all its work via the variable `each` and does not touch any of the member variables in `Customer` at all. This is another bad smell ("feature envy"). This suggests that the function should be moved from `Customer` to `Rental`.

Moving a function from one class to another is a refactoring referred to as the "Move Method". The process is:

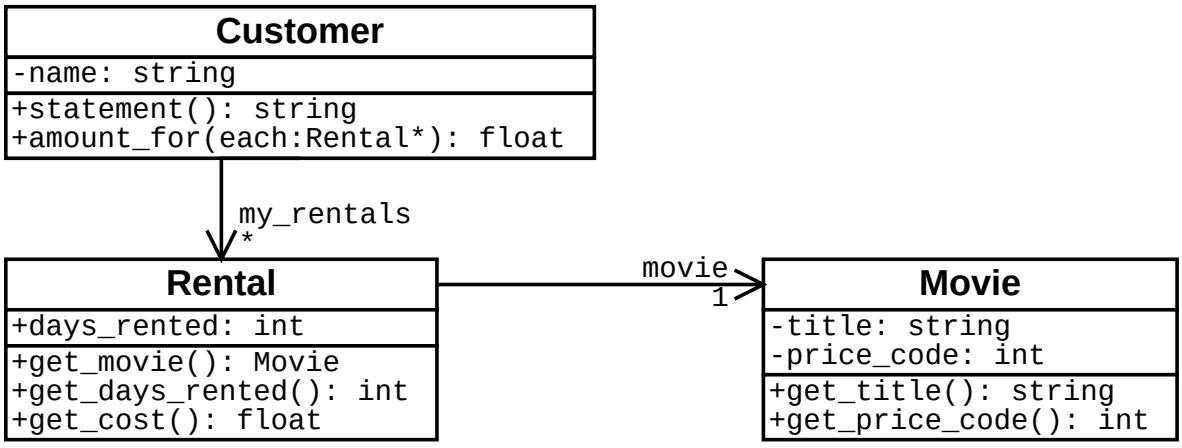
1. Declare a new function in the target class
2. Copy the code from the original into the new function.
3. All variables that were accessed via a pointer to the target class in the original function can now be accessed directly. In our example this means `each->get_movie()` can be replaced simply with `get_movie()`.

   If the code accesses some variables in the original class, they can be passed into the function as arguments or a route can be added to navigate back to the original class.
4. Compile the target class. This is a good chance to pick up any variables accesses which have not been changed correctly.
5. Replace the body of the original function with a simple call to the new function.
6. Compile and test.

We now have:

```
float Customer::amount_for(Rental* each){
  return each->get_cost();
}

float Rental::get_cost(){
  float this_amount=0;
  switch(get_movie()->get_price_code()){
    case Movie::REGULAR:
      this_amount += 2;
      if(get_days_rented() > 2){
        this_amount += (get_days_rented() - 2) * 1.5;
      }
      break;
    case Movie::NEW_RELEASE
      this_amount += get_days_rented() * 3;
      break;
    case Movie::CHILDRENS
      this amount += 1.5;
      if(get_days_rented() > 3){
        this_amount+=(get_days_rented()-3) * 1.5;
      }
      break;
  }
  return this_amount;
}
```



The `get_cost` function now uses information from `Movie` and `Rental` so at first sight, the function could live in either place. However, the presence of the `switch` statement based on the price code stored in `Movie` is an indicator that the function should be moved again so that we will have the opportunity to perform another refactoring. If we want to move the function to `Movie`, we shall have to take care of its use of `get_days_rented()` which belongs to rental by passing in that value as an argument to the moved function.

This gives us:

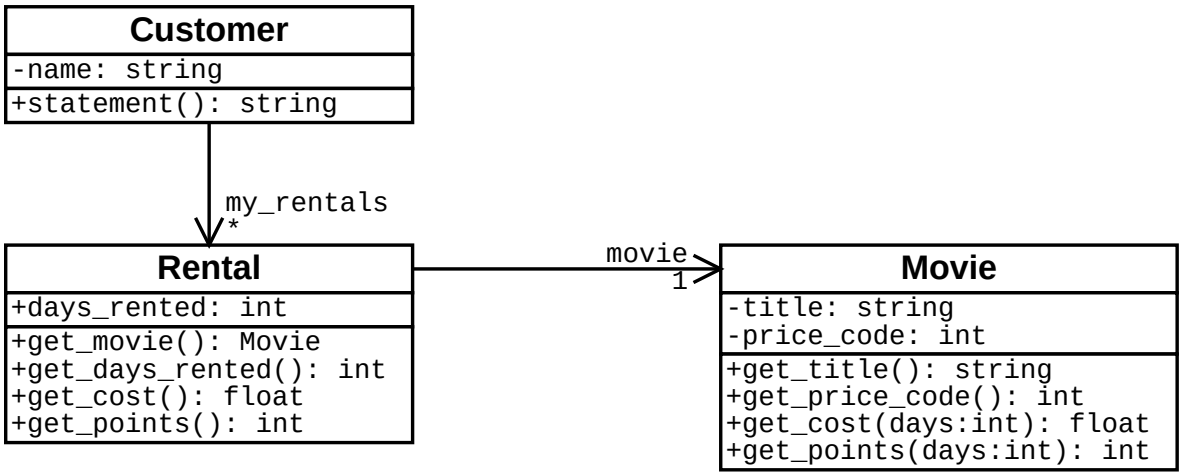
```
float Rental::get_cost(){
  return get_movie()->get_cost(get_days_rented());
}

float Movie::get_cost(int days){
  float this_amount=0;
  switch(get_price_code()){
    case REGULAR:
      this_amount += 2;
      if(days > 2){
        this_amount += (days-2) * 1.5;
      }
      break;
    case NEW_RELEASE
      this_amount += days * 3;
      break;
    case Movie::CHILDRENS
      this amount += 1.5;
      if(days > 3){
        this_amount+=(days-3) * 1.5;
      }
      break;
  }
  return this_amount;
}
```

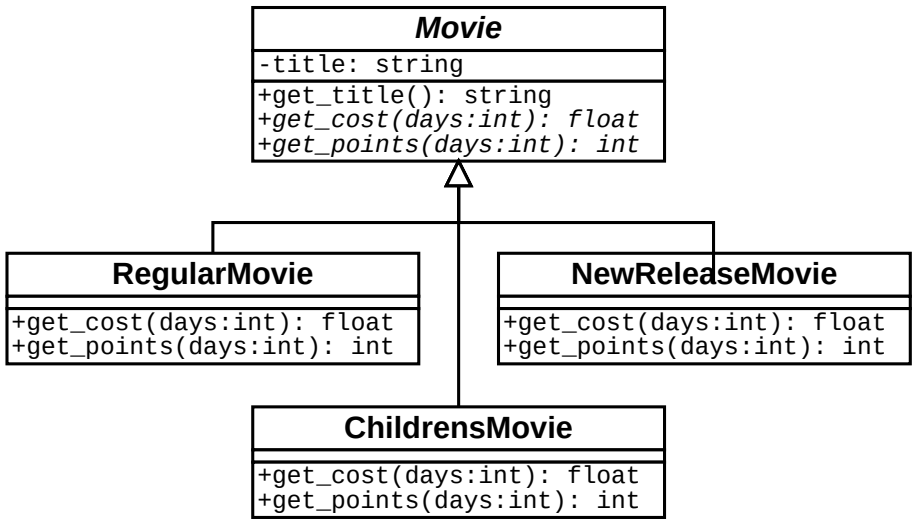
# Replacing `switch` with polymorphism

We can perform the same operation again on the code in the original function that computes frequent renter points, resulting in:

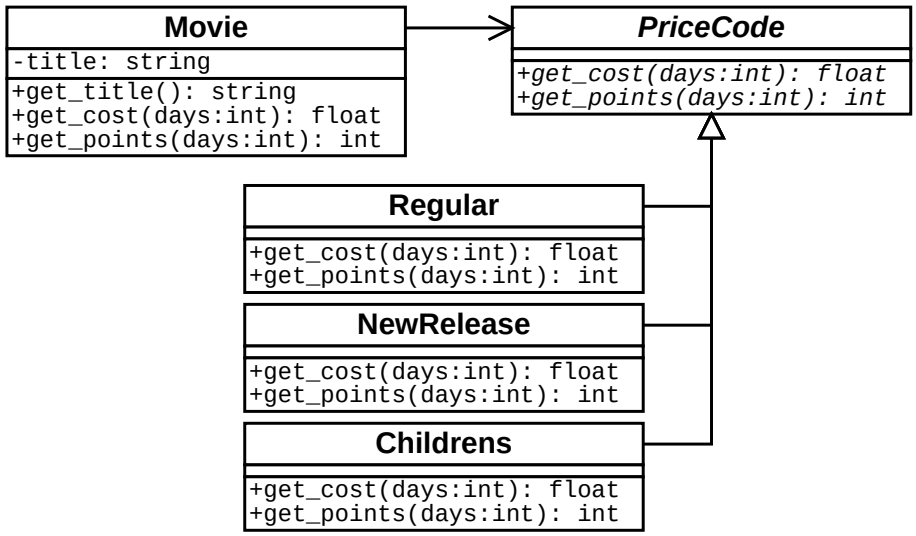
```
int Movie::get_points(int days){
  int points = 1;
  // add a bonus for two day new release rental
  if((get_price_code() == NEW_RELEASE) && (days>1)) {
     points++;
  }
  return points;
}
```




Both `get_cost()` and `get_points()` contain code which is conditional on the `price_code` member in `Movie`. This is another bad smell which can be eliminated by using polymorphism. There are two possible solutions to this; We can create several new subclasses of `Movie`, one for each movie type,




or we can use the state pattern:




In this case, the state pattern is the better design choice because it is likely that a `Movie` will change state from being a new release movie to a regular movie after it has been on the shelves for a year.