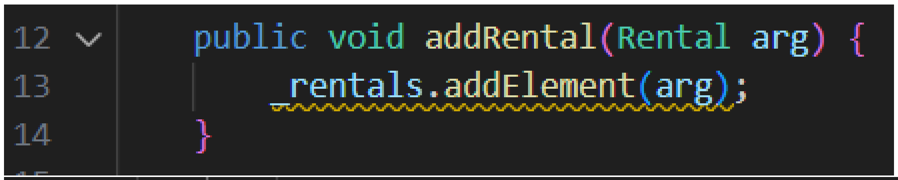
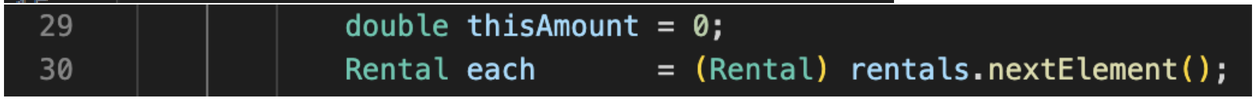
1. **Nondescriptive variable names and parameter signatures** (lines 12, 29-30 of Customer.java)



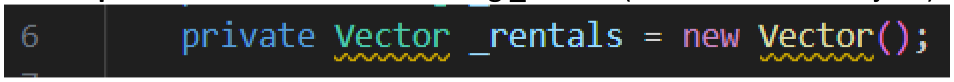
```
12      public void addRental(Rental arg) {
13          _rentals.addElement(arg);
14      }
```



```
29              double thisAmount = 0;
30              Rental each       = (Rental) rentals.nextElement();
```

Variable names like “thisAmount” and “each” are bad because they limit understandability. Similarly, “arg” is a poor choice for a variable in a function signature. Their meaning is not clear to a programmer from reading the code. There are other locations in the code with poor or misleading variable names.

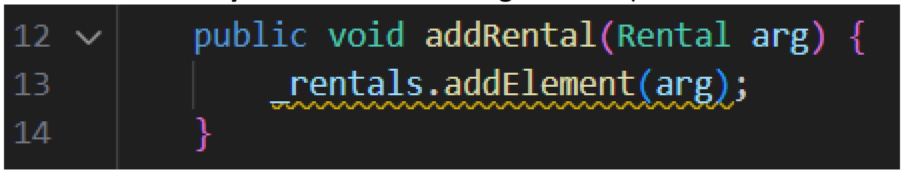
2. **No parametrization** when defining _rentals (line 6 of Customer.java)



```
6       private Vector _rentals = new Vector();
```

Visual Studio Code underlines this line for a good reason. The vector should parametrized as Vector<Rental> because _rentals is only supposed to contains Rentals, and it is not clear to a programmer than rental must only contains Rental objects. This would improve understandability and robustness.

3. **No validity checks** when adding a rental (lines 12-14 of Customer.java)

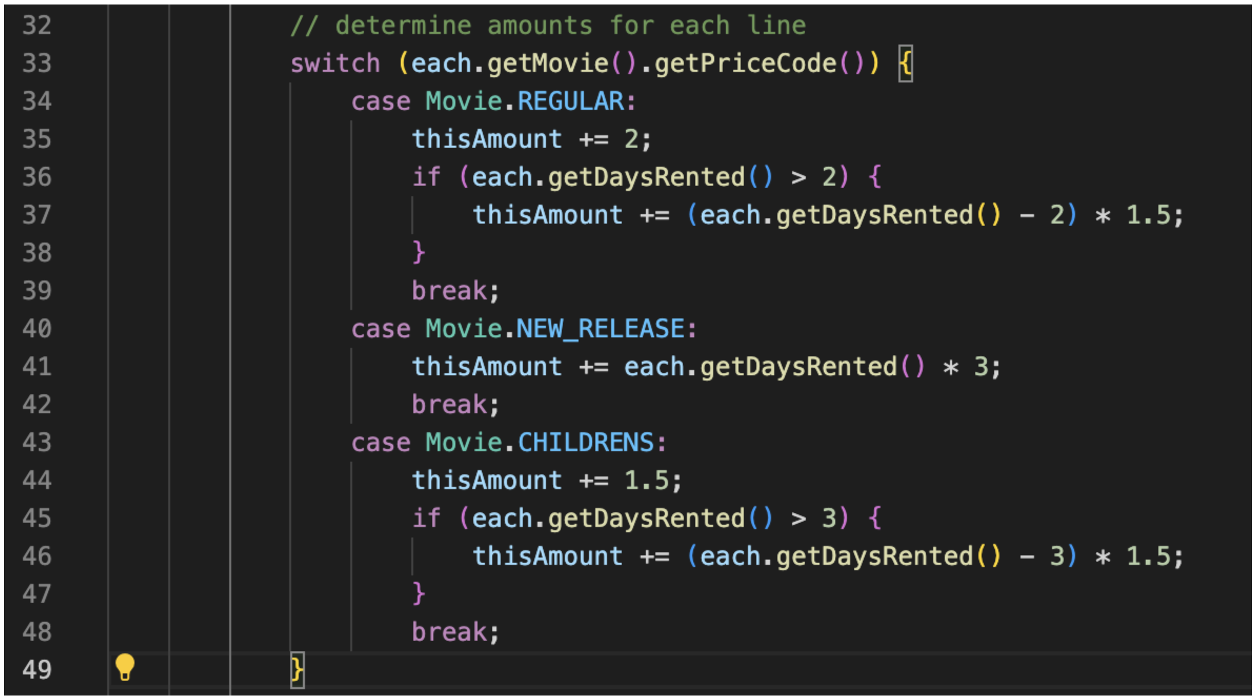


```
12      public void addRental(Rental arg) {
13          _rentals.addElement(arg);
14      }
```

There is no check that the new rental is valid, or more precisely that it is not null. If the rental is null, the statement() function (see below) would crash. There should be a check on this to improve robustness.

There are several smells in this section of code (lines 32-48 in Customer.java)

David Brodsky
CS 6359.001
Homework 1: Code Smells

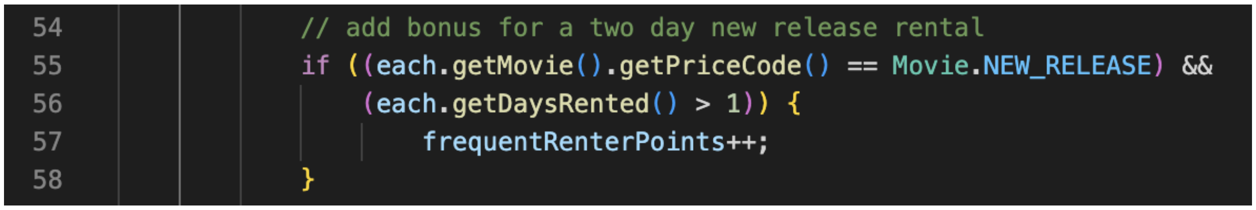


```java
            // determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (each.getDaysRented() > 2) {
                        thisAmount += (each.getDaysRented() - 2) * 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;
            }
```

4. **Switch statements** like this should not be used because they will likely be duplicated in code. This entire switch statement should be replaced with polymorphism. If more cases are introduced in the future, it will be difficult to maintain this switch statement and others. Therefore, this harms maintainability.
5. There is **no default behavior** for the switch case. If the movie type is not one of the ones listed above, the code would crash. Even though the price code should match one of the three, there should be a default case in case something goes wrong so the code doesn't crash. This harms the robustness of the software.
6. The calculations in the statement include **magic numbers** 1.5, 2, 3, the meaning of which are not clear. In particular, in the CHILDRENS case, it is not clear that the 1.5 numbers do NOT have the same meaning. That is not apparent to a programmer, so when one of the 1.5 numbers changes in the future, it will be easy to mistakenly change the wrong one. This impedes the understandability of the software.
7. Each case in the switch statement are **clone codes**, just with different numbers. The core logic is the same. If in the future the logic changes, a programmer will need to modify the code in at least 3 locations instead of just in 1. The 3 statements should be replaced with one statement, with the magic numbers as variables, to improve understandability and maintainability.
8. The Customer class is a **middleman** here. The logic in the switch statement is entirely dependent on the Movie class, so the statement belongs as a function in the Movie class. When the Movie class is modified, it is easier to maintain it if such functions that rely on Movie are in the Movie class rather than in another class.

There are multiple issues in this code fragment (lines 55-57 of Customer.java)

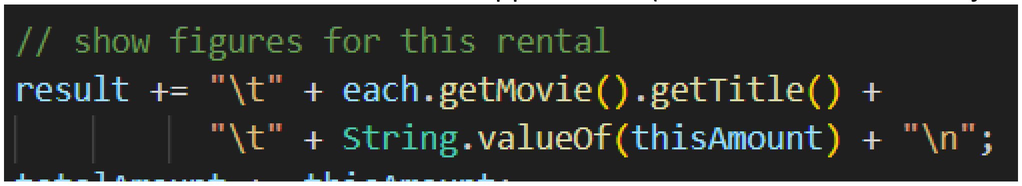
David Brodsky
CS 6359.001
Homework 1: Code Smells



```
54          // add bonus for a two day new release rental
55          if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
56              (each.getDaysRented() > 1)) {
57                  frequentRenterPoints++;
58          }
```

9. Line 55 contains message chaining. The enumerator object each calls the getMovie function, which returns an object that calls the getPriceCode function. If for some reason one of the functions returns a null object, there would be a Null Pointer Exception. Therefore, this should be implemented with only one function call.
10. The number 1 is a **magic number**. The bonus is supposed to be for a 2-day new release, but it's not clear that 1 = 2 – 1 logically. It would be better to replace 1 with a variable (e.g. bonusDays) that is more maintainable to edit. If the bonus period were to change, it would not be clear to a programmer to change the number 1.
11. As with the switch statement, the Customer class is a **middleman**. That entire fragment depends entirely on the Movie class, so it belongs in the Movie class as a function of its own. If changes are made to the Movie class, it will be much easier to maintain relevant functions if they are also in the Movie class.

There is a similar smell in the code snippet below (lines 61-62 of Customer.java)



```
// show figures for this rental
result += "\t" + each.getMovie().getTitle() +
          "\t" + String.valueOf(thisAmount) + "\n";
```

12. The Customer class is a **middleman** again. The format for printing out a movie rental details belongs in the Movie class instead of inside a function in the Customer class. This code is likely to be reused in some form, so for maintainability it should be in the Movie class. If modifications are made to the printout, or if new useful attributes are added to the Movie class, it will be hard to find this piece of code inside another class.

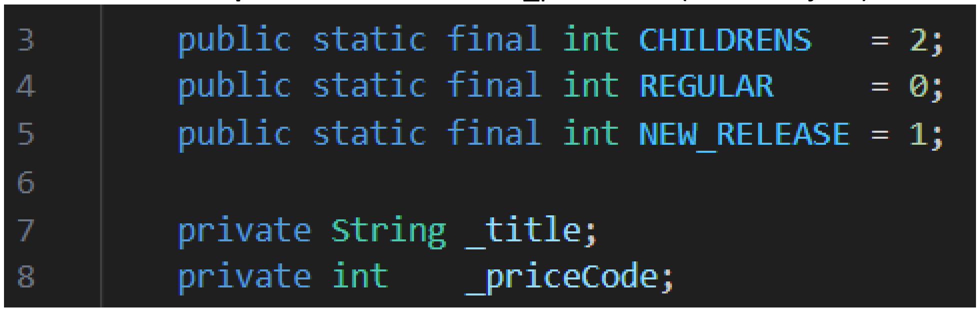
There are also more general issues with the Customer class.

13. The statement() function is **too long**. It performs multiple functionalities at once—computing the amount, number of points, and receipts for a customer—without calling any other functions. The statement() function could be split into at least 2-3 smaller functions, which could then be called in the statement() function. For example, having functions called getAmount(), getPoints(), etc. would be better. That would make the code more understandable, more maintainable, and easier to test.
14. The statement() function is **inefficient** because it recomputes all the values every time it is called. It would be more efficient to store the user's total amount, points, and even receipt as class attributes and update them every time a rental is added (in addRental()). In a real-life system, iterating through every movie a customer has rented and recomputing things would be a waste of time.

David Brodsky
CS 6359.001
Homework 1: Code Smells

Moving on to the other files:

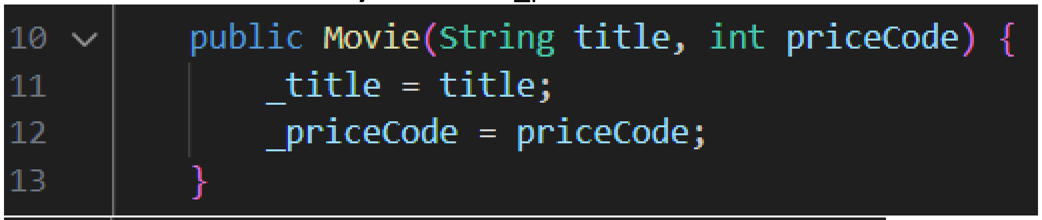
15. **Nondescriptive attribute name**: _priceCode (in Movie.java)



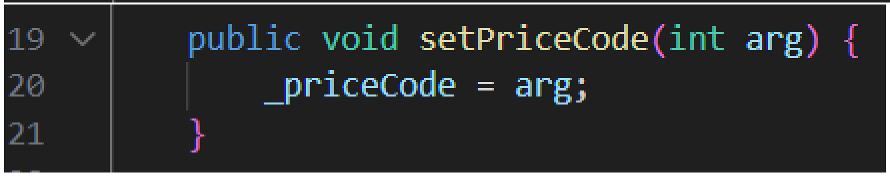
```
    public static final int CHILDRENS   = 2;
    public static final int REGULAR     = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int    _priceCode;
```

It is not obvious that a) _priceCode must be CHILDRENS, REGULAR, or NEW_RELEASE and b) _priceCode represents different categories of movies. That is simply not obvious from the name or the rest of the code inside Movie.java. _priceCode should be replaced with a more descriptive name and/or an explanation that it is related to the static variables above, because a programmer reading this code could easily make a mistake or misunderstand the purpose of _priceCode.

16. There are **no validity checks** on _priceCode in the constructor and in setPriceCode().



```
    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }
```



```
    public void setPriceCode(int arg) {
        _priceCode = arg;
    }
```

As becomes clear from reading the statement function, _priceCode must be either CHILDRENS, REGULAR, or NEW_RELEASE. In the two snippets above, there are no checks to ensure that _priceCode is one of those three values. This is a code smell because it allows a programmer to accidentally set _priceCode to an invalid value, which would cause serious issues elsewhere in the software.

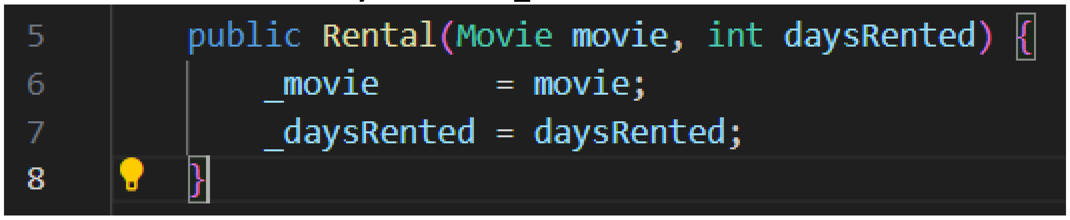
17. arg is a **nondescriptive parameter signature.** This harms understandability because a programmer will not understand what "arg" is supposed to represent.
18. Movie is a **data class**. It only contains a constructor and getter and setter functions. It contains no behaviors. This is bad because classes should contain behaviors instead of just storing data. In this case, it would clearly be best to move most of the logic from the statement() function to the Movie class.

David Brodsky
CS 6359.001
Homework 1: Code Smells

19. Rental is also a **data class**. It only contains a constructor and getter functions. It should contain behaviors instead of just storing data.

Additionally, there is another smell in the Rental.java file.

20. There are **no validity checks** on _movie in the constructor.



```
    public Rental(Movie movie, int daysRented) {
        _movie      = movie;
        _daysRented = daysRented;
    }
```

There is no check that movie is not null. In the statement() function, there is a message chain that calls getMovie.getTitle(). If movie is null, the program would crash.