Inf2-SEPP 2025-26

Tutorial 5 (Week 8)

Notes on answers

1 Task 1: Coffee calculator tool

1.1 Task 1.1: High quality code, code review

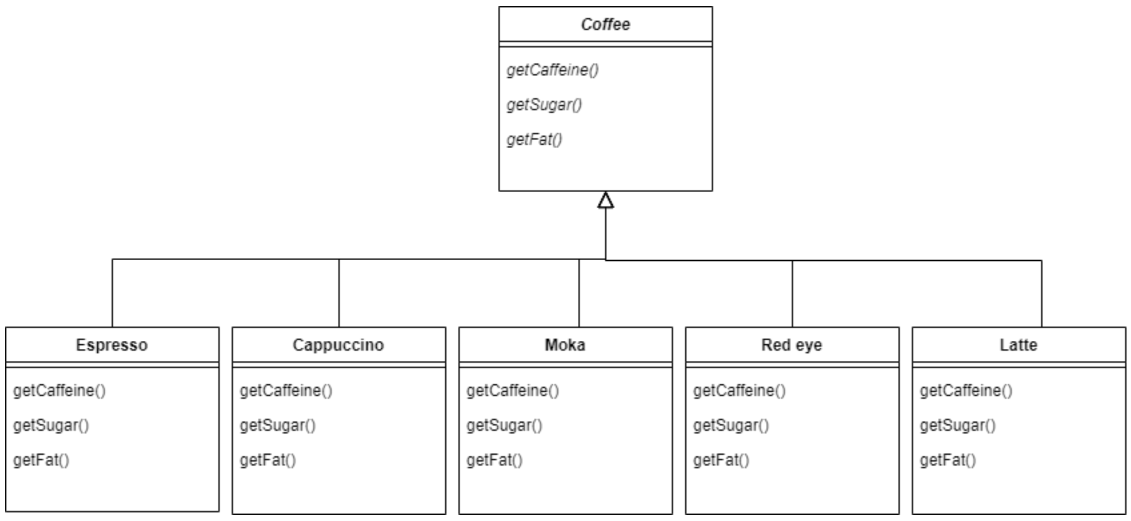
1. Ways of improving the code according to the principles of good code:

   - Bracketing put consistently either as paired up or starting after method name/condition, ending at start of end line
   - Indentation used for the method body (x1), each case statement (x2), each command in the case (x3); Made consistent
   - Field names made more intuitive (e.g. glass_size and not gc)
   - Spacing consistent for operations (before/after operators and brackets) and after ':' of case
   - Comments not used for obvious lines of code
   - Commented unused code removed
   - Method names using the camel case convention
   - Using variables to explain and reuse magic numbers
   - If `espresso` and other coffee names are intended to be constants, using snake upper case for them, e.g. `ESPRESSO` and not `espresso`

2. You are highly recommended to try out writing this code.

1.2 Task 1.2: Refactoring

1. The code can be refactored by replacing the switch/case conditional with polymorphism. Classes Espresso, Cappuccinno, Moka and Red Eye could extend an abstract Coffee class, and each would override its abstract getCaffeine method, rather than have all the different methods for getting the caffeine in a single Coffee class. Thus, depending on which class an object holds a reference to, calling getCaffeine in it would call the overriding such method from the inheriting class. New classes for other coffee



types can be easily added. Moreover, getSugar and getFat methods could also be abstract methods in the Coffee abstract class and inherited and overriden by all of the coffee type classes. It makes sense for Coffee and these methods to be abstract in it, because each coffee would normally have different caffeine, fat and sugar levels given by the differences in their recipes, and so we need them all to override these methods and not re-use an inherited regular one. Common characteristics of coffees could be stored in attributes of this abstract class, and inherited by the subclasses. Any other methods which may work similarly in subclasses can be defined as regular methods in the abstract class and inherited if reused (or overriden if not) in the subclasses. We assume that attributes and other methods could be added to the Coffee class, which justifies this class being abstract. If this is not needed, Coffee could instead be an interface that is realised by the classes representing different types of coffee.

2. A class diagram of this could look as shown above.

2 Task 2: Movie rental store [1]

1. Some code smells in the statement() method's code and the code indicated by the class diagram, together with reasons why they are problematic, are:

   - statement() is a long method, doing several things at once- difficult to understand, debug, maintain
   - statement() contains message chains e.g. in each.getDisc().getMovie().getPriceCode()- increased coupling (associations between the current class Customer and classes in the chain), makes system rigid, makes classes difficult to test independently
   - Feature envy: several parts of the statement() methods are concerned only with using functionality from other classes: the part on determining the amounts for each line, and adding frequent renter points, which are only using functionality from the Rental class- reduced cohesion

[1] inspired from Martin Fowler's refactoring example

- Data classes: Rental, Disc and Movie only contain attributes and getters- reduced usefulness of these classes, to which some functionality should be moved.

2. A naive solution would be adding an htmlstatement() method, but then this method would duplicate most of the code from statement() (duplicated code smell), which reduces maintainability and increases the chance to introduce faults.

3. The following are the most important refactorings to do:

   - Extracting the amount calculation (switch statement) to a separate method in the Customer class. This reduces the length of the statement() method (addresses large method). As we do so, making non-modified variables into parameters, and the modified variable as something local that is returned.
   - Renaming variables in the new method to make them more intuitive (makes code higher quality, improves understandability and maintainability).
   - Moving the new method to the Rental class as it only uses information from the rental and associated classes, and none from the customer (addresses feature envy code smell, message chains code smell, data class code smell).
   - Doing all of the above for frequent renter points
   - Moving the rental calculation to the Movie class, because the switch acts on an object of Movie, taking the days rented as a parameter (addresses feature envy code smell, message chains code smell, data class code smell)

   The above changes also mean that adding an htmlstatement() method to Customer would result in a much shorter method with much less duplicated code with statement() (reduced duplicated code smell). The functionality of these methods could further be extracted into 3 methods, used by each with different parameters: one for the header of the statement, one for its body and one for its footer. This would reduce code duplication even further.

   For a full description of the changes made, together with code, see Martin Fowler's refactoring example (the example there talked of tapes, here updated for more modern times).

4. See above link for code. Changing the class diagram involves reverse-engineering from the code. You are highly recommended to try to write this code and refactor it.