By: Sergio Puertas Pérez and Xabier Xoán Carricoba Muñoa

MEMORIAL, DS DESIGN ASSIGNMENT, EXERCISE 1

The exercise asked for a program to simulate efficiently an online shopping website. Our implementation consists of a series of steps (Shoppingcart, chekout, payment and, depending on previous states, completed or cancelled).
These states are created using an interface "Step" which we will be changing during the program's execution depending on the behaviour of the user.

HOW DOES THE PROGRAM WORK?
The idea we had is to create an interface Step, and a class Ord, that manages the user input. The states would implement the Step interface, and there would be some operations, op1 and op2, that would do different things depending on the state in which the user is. The methods will be overriden to those classes so we can simulate what a real shopping website is: each step also has certain operations that you can or cannot perform, and this way you don't need to fill the code with loads of different methods.

The use of the interface which holds the steps makes that, in the future, new steps can be created. These steps will only need to be defined as a class that implements the step interface.

The flow of the program can be defined as follows: you start in the shoppingcart state, in which you can add and remove items from your cart(disguised as op1 and op2), then you can advance to checkout phase (using the method next), in which you also have different operations to perform (one of which is to go back to the cart, privilege that only that step has), then if wanted you can go to payment phase where the user action almost finishes, as it is posible to go to step cancelled if the user decides so. Final states completed and cancelled don't implement methods op1 and op2, and neither does it payment, as they are more of a transition state.

In addition, we created the Log, to have every action performed in the order registered in there.

WHICH PRINCPLES DID WE FOLLOW?
To create the interface, the first thing to do is to make the UniqueInstance attribute of type Step, as specified in the theoretical lessons. With this method, we can compare and use the different steps without creating any of them in the Ord class, just calling this method. The same goes to the method next, that lets us know the next step of the shopping within the steps themselves, acting as a independent entity.

Moreover, the "Inverted Dependence" principle is also present on our project since we always use the interface step to take care of the operations, states and decisions the user must take, instead of using a concrete classes.

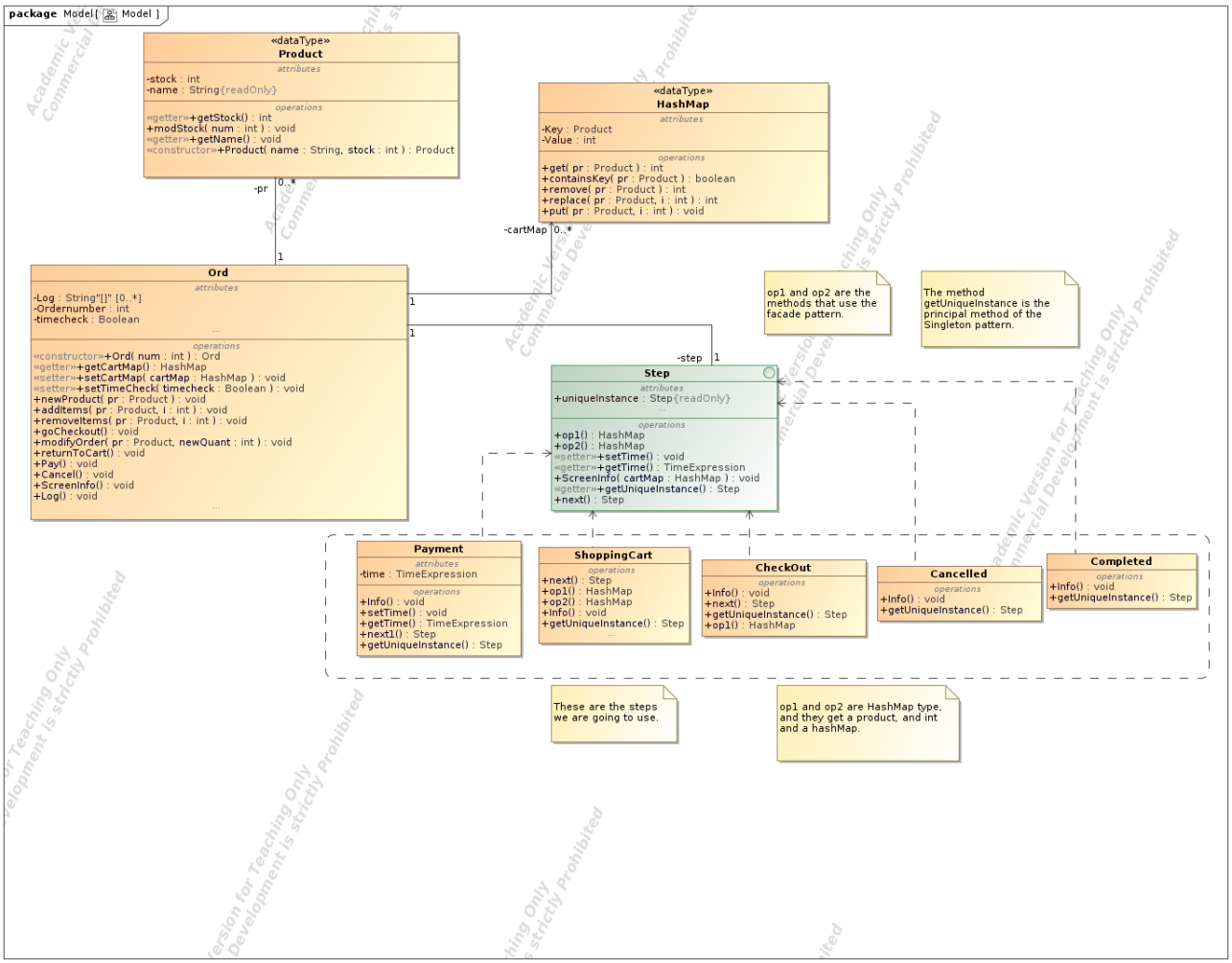
Finally, another SOLID principle from which we take advantage is the "Open-Closed" principle. As it was previously explained, the program may be expanded by adding new steps that implement the delivery process (delivered, lost in transit...), so we can create new steps, in order to fulfill our needs (Delivered, Lost, Going...), and they will never modify the rest of our classes, just expand them.

WHICH PATTERNS DID WE USE?
As we said earlier, with the UniqueInstance methods, we use the Singleton pattern, that basically applies that method.

We also use the Façade Pattern, as in the Step Interface there are methods called op1 and op2, that don't have a specific use. It is decided by the classes that implement them. op1 in Shopping Cart is to add items to the cart, but in Checkout, it's used to modify the quantities of the products already introduced in the cart, as an example. We don't take a lot of advantage from this pattern, but when the process is expanded, probably we would be able to maximize its potential.

But the main pattern used is the State pattern. Every class that implements Step is a state, and they perform as that. In the Ord, there is an object Step, and during the shopping it takes different values. It changes its value whenever we want to advance or return in the process. To change its value, we use the getUniqueInstance method, present in every state class. In its actual version, the last steps are Confirmed or Cancelled, but the idea would be to add every step related with the delivery of the order. For example, we could create Delivered, Lost and Going.
UML DIAGRAMS

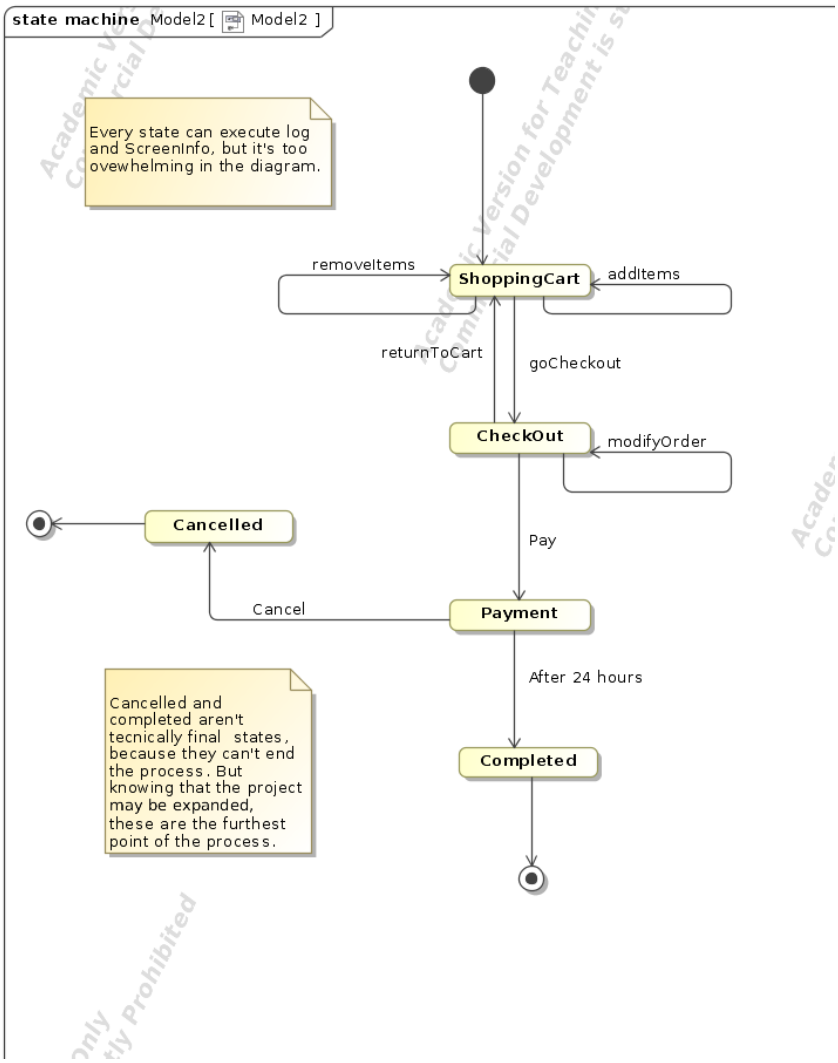
CLASS DIAGRAM



As you can see, there are not many classes, just some steps and the basic information (Product and Ord). We try to avoid the “God Class”, but we need to have a class that manages the pathing of the program. In our case, it’s the Ord class.

By: Sergio Puertas Pérez and Xabier Xoán Carricoba Muñoa

STATE DIAGRAM



The state diagram is quite simple, as the only thing we worried about is the states, and the transitions between them. We don't go in depth into each of the calls.

In a nutshell, our code performs the online shopping task with ease, and without taking any sideroads. Also, it takes benefit the different Patterns and design principles used during the coding.