Victor Nathanael Badillo Aldama

Pablo Legide Vidal

E1 REPORT

DESIGN PATTERNS FOLLOWED

- **State pattern:** the reasoning behind this election is, that we have some common methods such as *screenInfo*, that need to be implemented differently depending on the step in which the **Order** (playing the context role) is throughout the online shopping system. Said methods are contained by the **OrderState** interface (playing the State role) and implemented differently for each step of an order, which can be: **ShoppingCart**, **CheckOut**, **Payment**, **Completed** or **Cancelled** (playing the concrete States role).
  Using this pattern for this exercise we get some advantages such as making it easy to insert new states to the system by just adding a new class and making state transitions between steps explicit, by using methods such as *changeState* or *stepBack*.

- **Singleton pattern:** by using it, it is ensured that the desired class has only one instance, which is what we want for each step of the online shopping system.
  The initialization method chosen for the singletons will be the early one, so the instance is created when the class is loaded by the compiler, ensuring the access to the same instance through the whole program. By doing this, **ShoppingCart**, **CheckOut**, **Payment**, **Completed** and **Cancelled** will be unique instances shared by all the different orders, which makes more sense than creating the same instance multiple times for each order.
  We will use the different singletons in the above-mentioned state pattern as if they were elements of an enumerated type, thus, minimizing the problems this design pattern may have.

DESIGN PRINCIPLES FOLLOWED

- **Open Closed Principle:** followed in the **Order** class by delegating most of its methods to the **OrderState** class which is an interface with said methods. These methods, such as *addItem* or *printPhase* will be redefined for each concrete state, so they satisfy its required function depending on the step of the shopping system in which they currently are, for example the printPhase method behaves different in each one of the steps.
  By converting each concrete state such as: **ShoppingCart**, **CheckOut**, **Payment**, **Completed** or **Cancelled** into classes, we allow the future modification, addition, or deletion of the different steps the online shopping system needs, without changing the source code and without limitation.

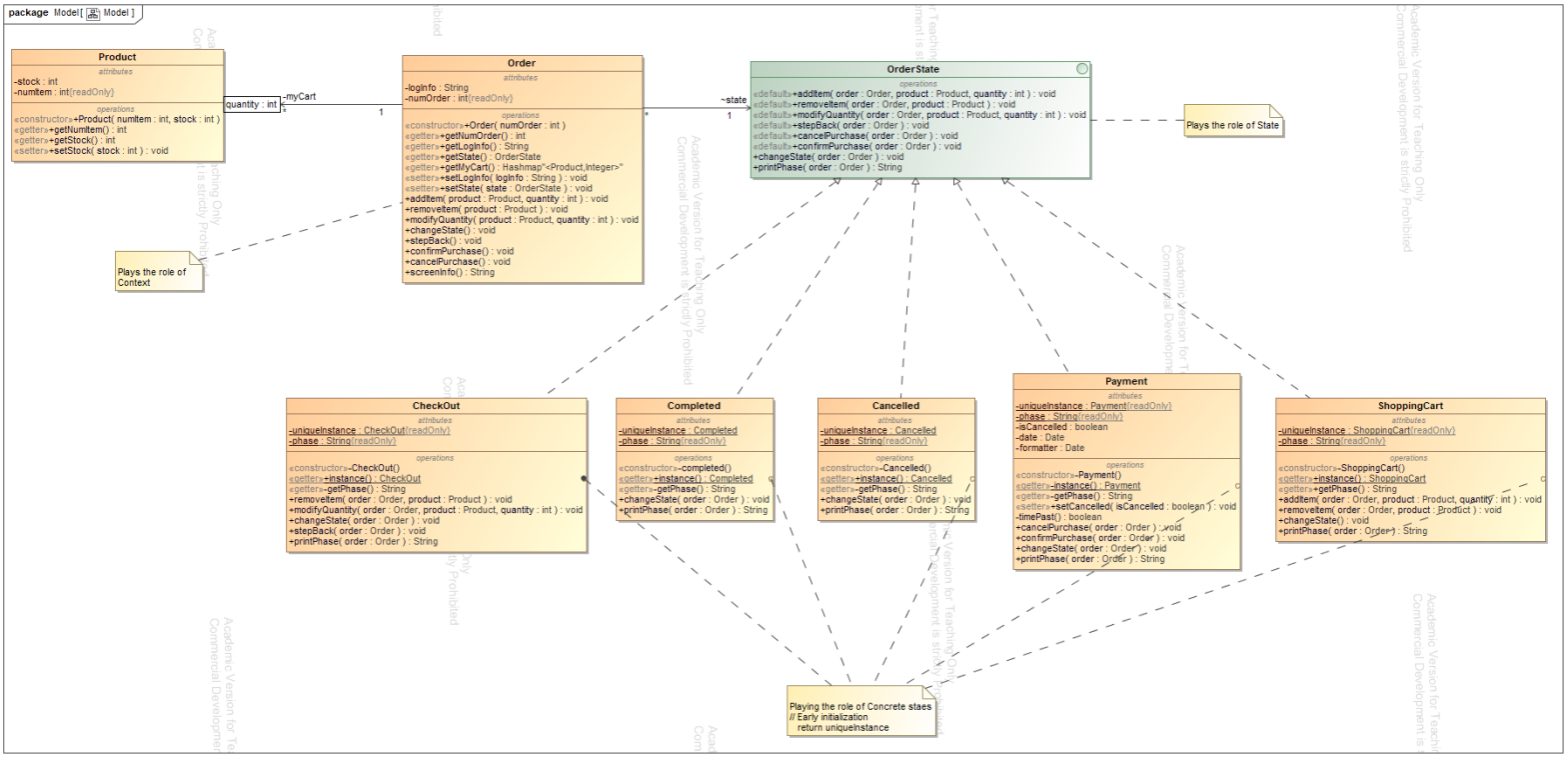
- **Liskov substitution Principle**: followed by the **OrderState** class and all the classes that implement this interface: **ShoppingCart**, **CheckOut**, **Payment**, **Completed** and **Cancelled**, when they override a method contained by the interface **OrderState** redefining it.
  As it can be seen in the *printPhase* method the screen info can be printed for any state even if it modifies the *printPhase* behavior of the **OrderState** class, without contradicting the LSP as any client of the online shopping system can expect what the screen will print in each one of the steps, this means that the redefinition of the method meets the principle of subcontracting.

- **Dependency Inversion Principle:** which is met when creating orders from the **Order** class, which depend on the **OrderState** interface so that they can use the methods from this interface in whichever state they are, even if its behavior depends on this state. This is because when defining the order methods, we do not specify the use of a single and concrete state but a state in general, so any order in any state may use them. For example the method *removeItem* from the order class is defined as *state.removeItem,* so when it is called it will use the state in which that order currently which can be any of them.
  We can also see dependency injections, such as the Constructor injection in methods such as the *Order* constructor from **Order** class which save saves the reference to the passed-in numOrder inside this Order, or setter injection with the *setLogInfo* from Order class.

- **Encapsulate What Varies Principle:** followed by the different classes: **ShoppingCart**, **CheckOut**, **Payment**, **Completed** and **Cancelled** that represent the different online shopping system steps and implement the **OrderState** interface.

This classes are separated from the things which always stays the same which are the orders from the **Order** class because they are the ones which are the most likely to be modified in a future for example, by changing the info that the *printPhase* method displays or making it possible to change the time in which orders go from Payment to Completed state to 12 hours instead of 24 or even by creating new steps and deleting the existing ones.

CLASS DIAGRAM:



DYNAMIC DIAGRAM: state machine diagram