Faculty of Computers, Informatics and Microelectronics
Technical University of Moldova

IPP

Lab#3

Author:

Valeria BEGA

Supervisor:

Anastasia SERSUN

Chisinau 2017

1. Objective

Behavioral Patterns

Implementing 5 chosen Structural Patterns:

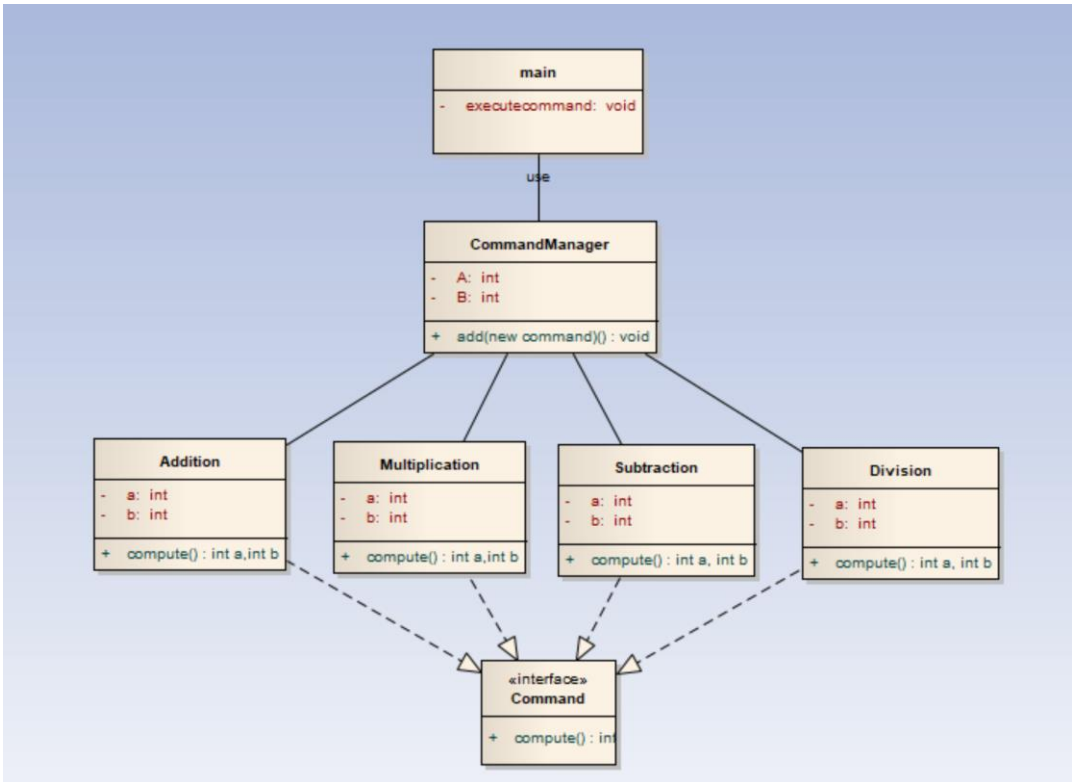
Command, Observer, Strategy, Mediator, Interpreter

Command

- Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- Promote "invocation of a method on an object" to full object status
- An object-oriented callback

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Here we have Command which is an interface and basically has declared only the general method. Then there's the operations, which implement this interface and have their concrete methods. Through the command manager the user can issue different commands without knowing anything about the operations being requested. So, all the logic is store inside the CommandManager. This pattern decouples the the object that invokes the operation from the one that knows how to implement it. It acts as a black box.



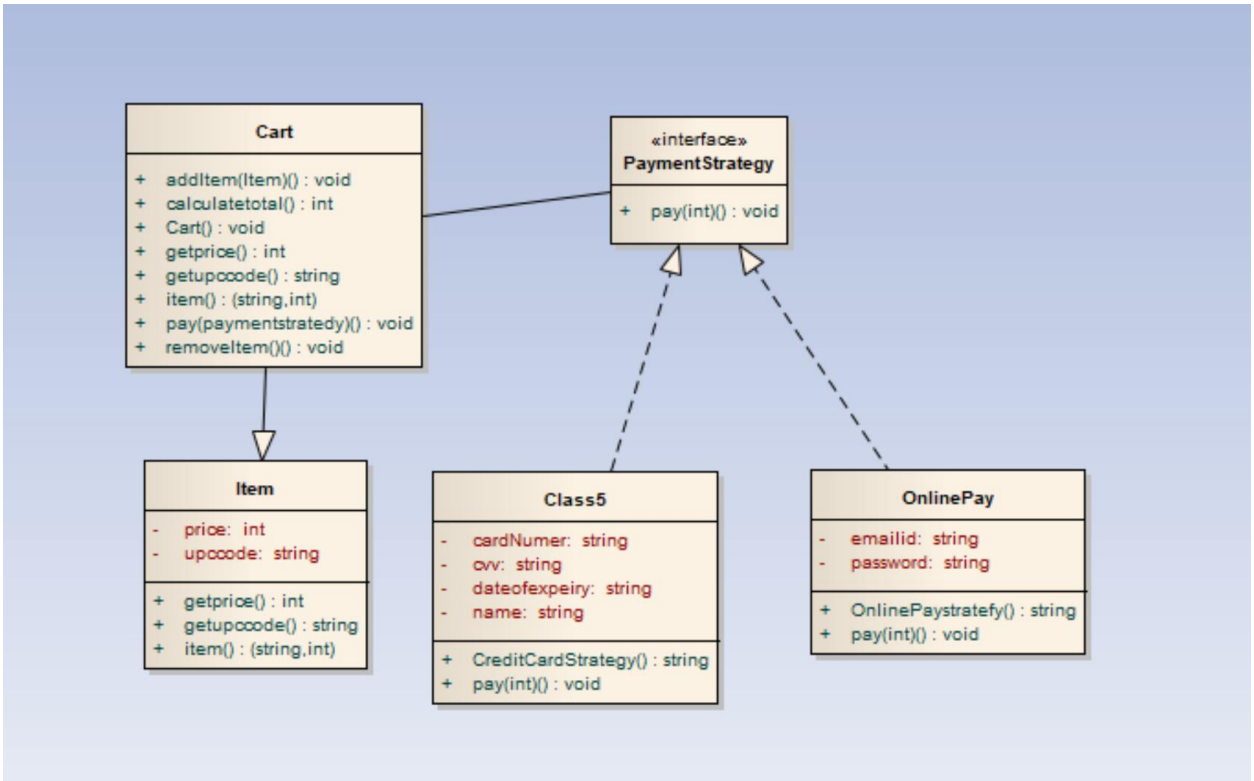
Strategy

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

One of the dominant strategies of object-oriented design is the "open-closed principle".

Here I implemented a situation at a shop place, we have a cart where we add items, get their prices and compute the general sum. We have the item, with the prices and the codes defined. And then we have two strategies. Paying by card and online paying. The client chooses which strategy to use.

Pattern is useful when we have multiple strategies that have to be applied to different specific tasks, so we want the app to be flexible and we want to choose the algorithm at the run time.



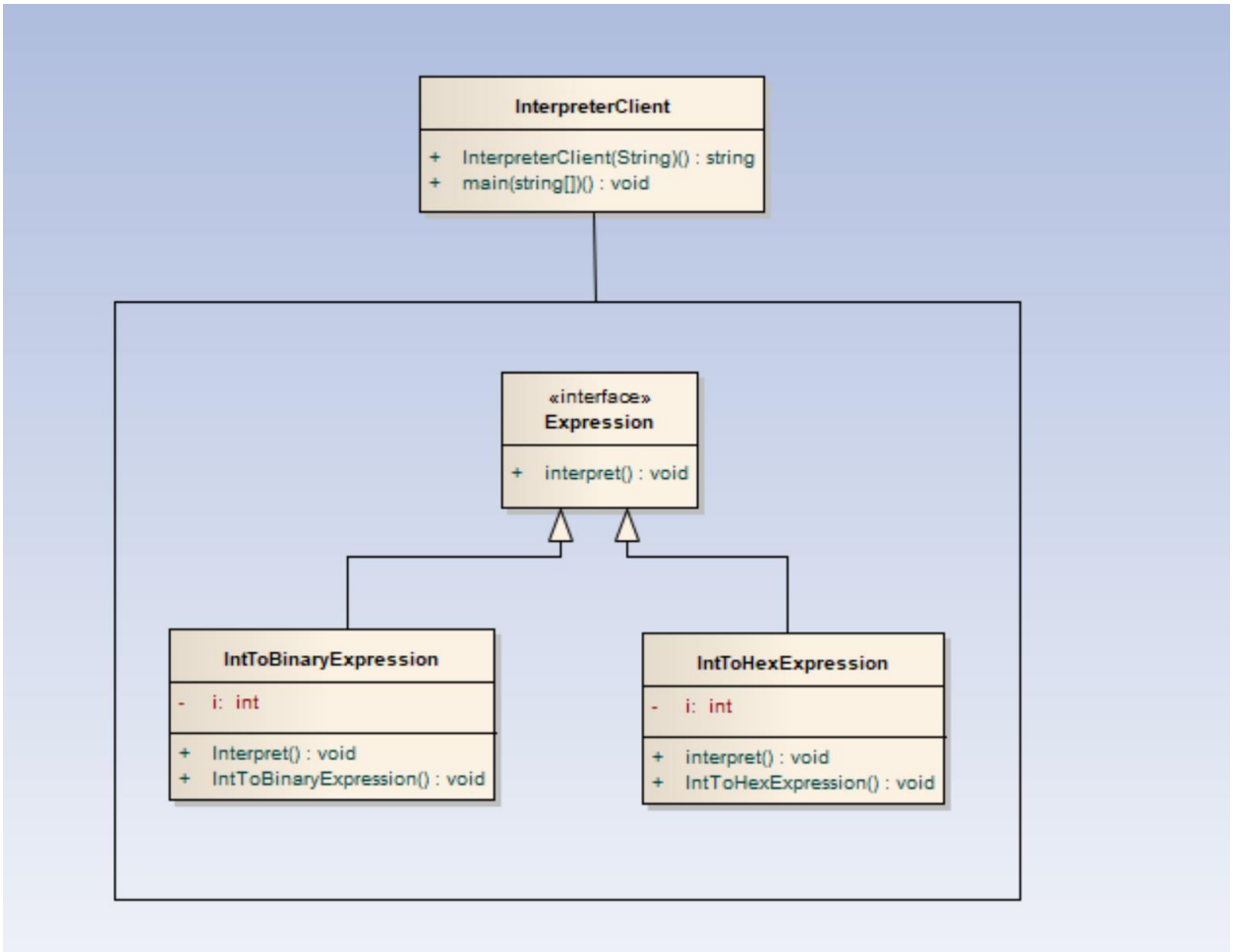
Interpreter

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

Here we have a situation, where the user want to transform int to binary and hex. So we have and expression interface and two implementations of it.

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.



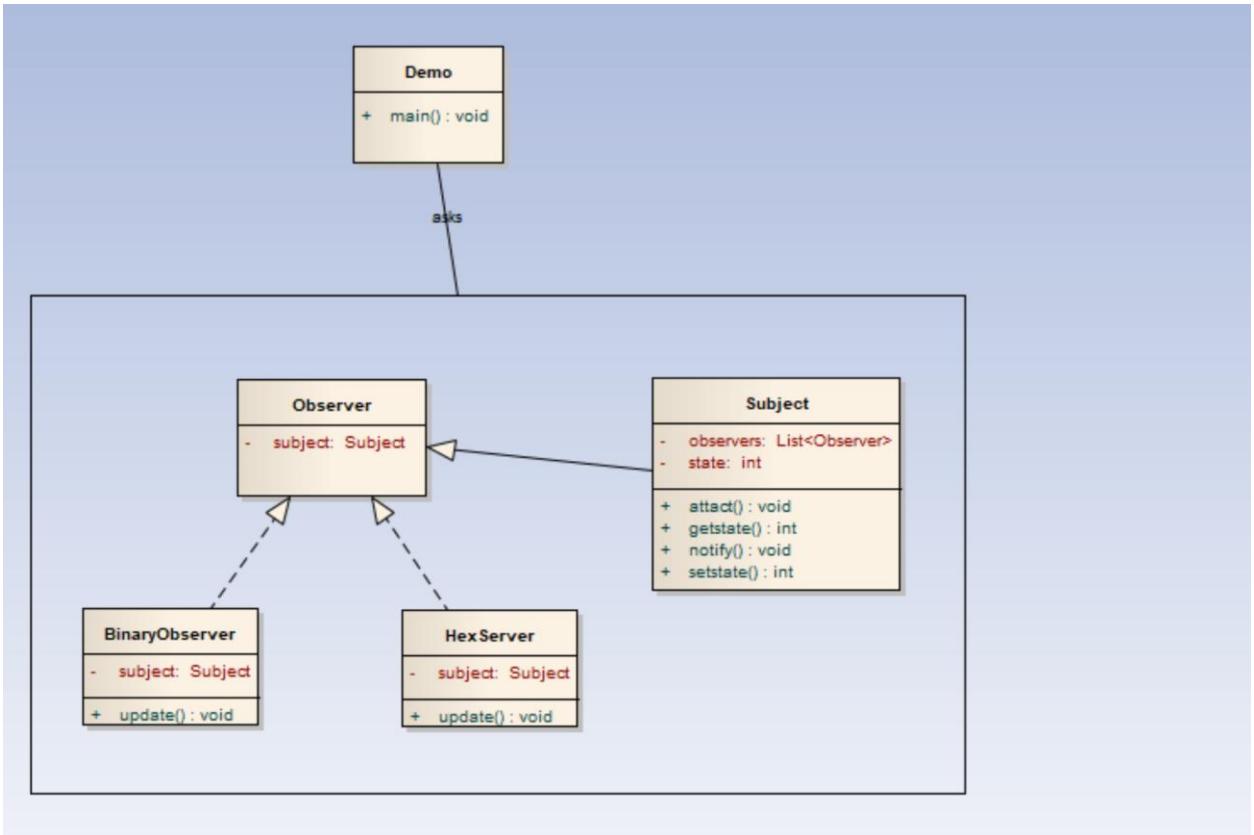
Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The "View" part of Model-View-Controller.

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Here we also have the situation with the int transformation, but the thing is that we update and notify them in the client part. So, whenever the subject changes, observers change too with the new changes.

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically.



Mediator

- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to "full object status".

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

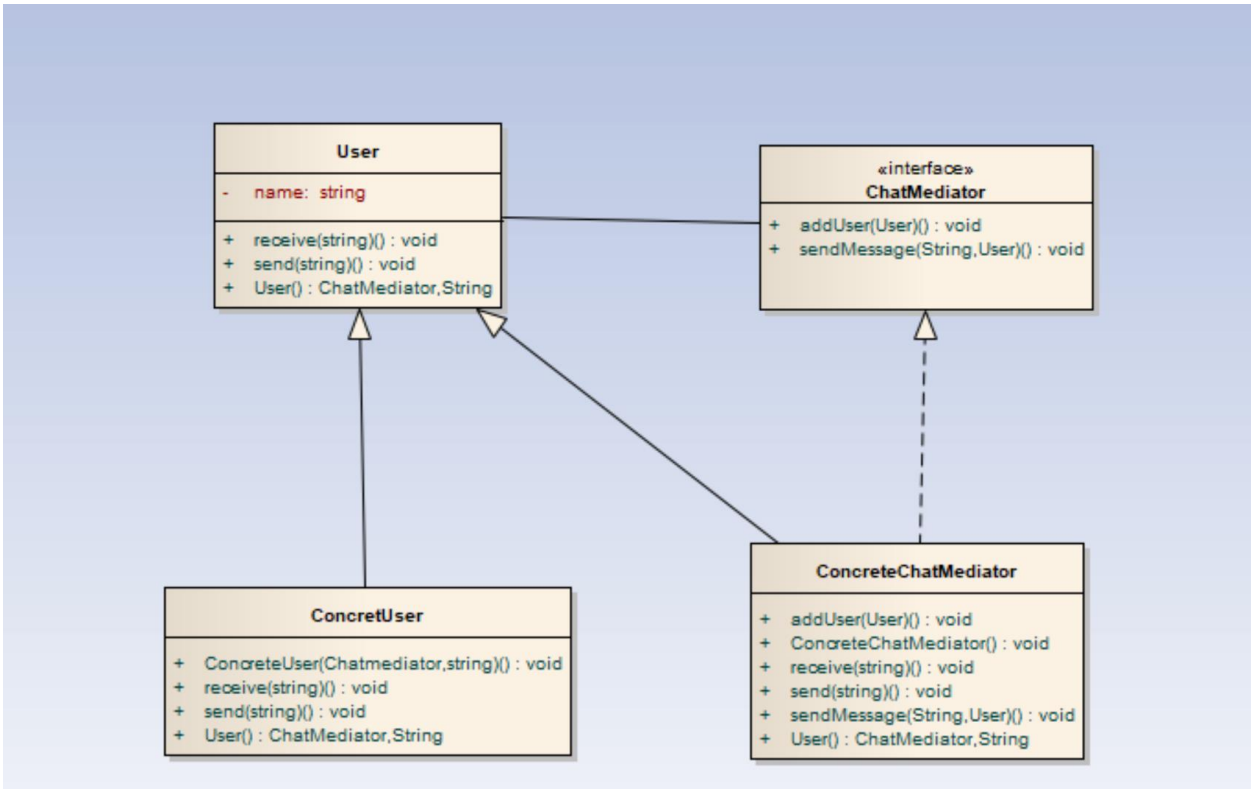
Here, we have a chatgroup, where multiple users interact between each other, so they are usually tightly coupled. Mediator helps improving loose coupling and help communications between these

objects(the users). There's an interface that provides the definition of the mediator, and then there's a concrete implementation. The message sent by an user should be received by every user in the group.

Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.

Java Message Service (JMS) uses Mediator pattern along with Observer pattern to allow applications to subscribe and publish data to other applications.

We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.



Conclusion:

For the code, see the link here [link]

Elaborating there laboratories I learned about the structural patterns, which are kinda more difficult than the creational ones to understand. Got to see again that these patterns exist to improve the reusability and the easiness to write code. Also, doing the diagrams for each pattern, made me remember better their structure.