Sales App Documentation

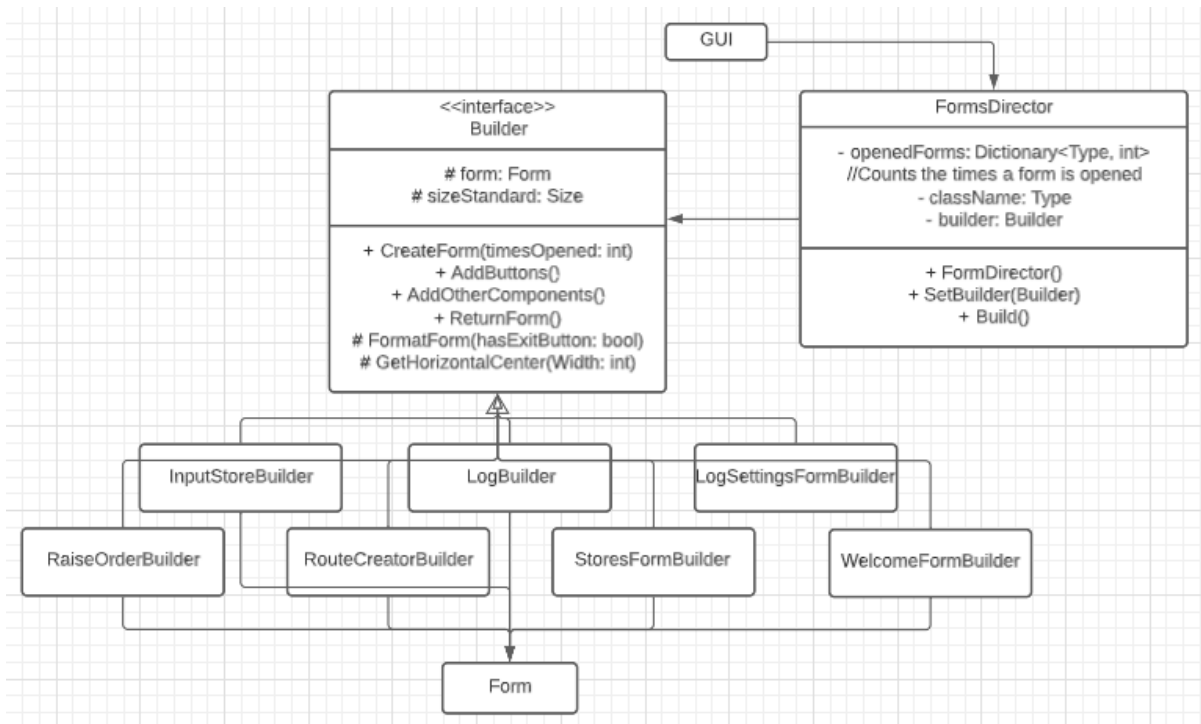> See the problem description!

Design Pattern

The patterns implemented in this project are the following ones:

Creational

💡 Builder

This pattern was used to generate and setup every form, the client of this pattern was the class `GUI` and it was in charged of managing the execution of every page.
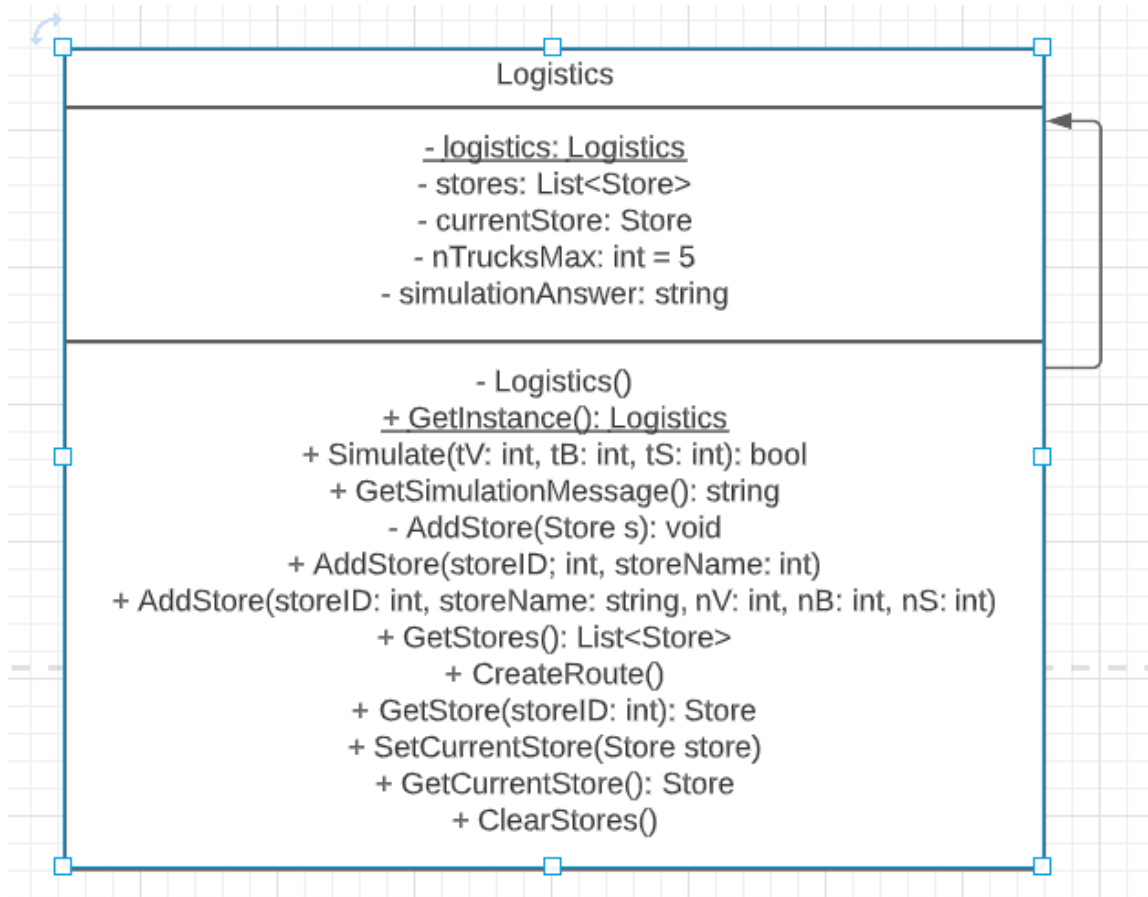
> 💡 <u>Singleton</u>

Two singletons were implemented in the solution:
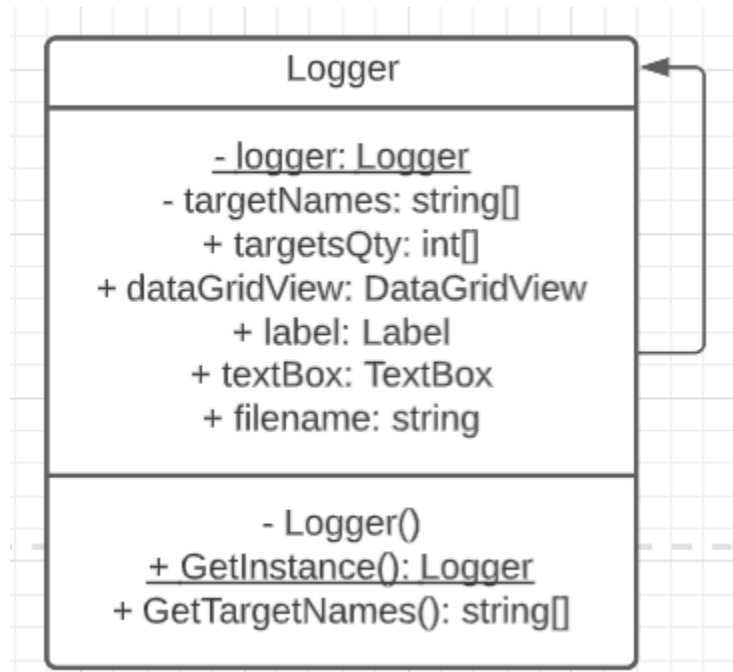
- Logistics

  A singleton was useful in this case because it is a class able to deal with lazy initialization, and a single instance with global access was going to be needed.

  This class is responsible for all the logic related to `Store` management, simulation and creation of routes.

Logistics

- logistics: Logistics
- stores: List<Store>
- currentStore: Store
- nTrucksMax: int = 5
- simulationAnswer: string

- Logistics()
+ GetInstance(): Logistics
+ Simulate(tV: int, tB: int, tS: int): bool
+ GetSimulationMessage(): string
- AddStore(Store s): void
+ AddStore(storeID; int, storeName: int)
+ AddStore(storeID: int, storeName: string, nV: int, nB: int, nS: int)
+ GetStores(): List<Store>
+ CreateRoute()
+ GetStore(storeID: int): Store
+ SetCurrentStore(Store store)
+ GetCurrentStore(): Store
+ ClearStores()

- Logger

  This class is in charged of writing the operations made
  during the execution of the program and contains useful
  information. It also provides access to the containers that
 are going to have the log information.

```
                    Logger

        - logger: Logger
        - targetNames: string[]
        + targetsQty: int[]
      + dataGridView: DataGridView
         + label: Label
        + textBox: TextBox
        + filename: string

        - Logger()
      + GetInstance(): Logger
    + GetTargetNames(): string[]
```

Structural

> 💡 Adapter

This class works in collaboration with the `Logger` class, because that Singleton plays the role of the adaptee, containing the necessary targets for the logging information, and the adapter classes provide the specific implementation needed to register into different containers.
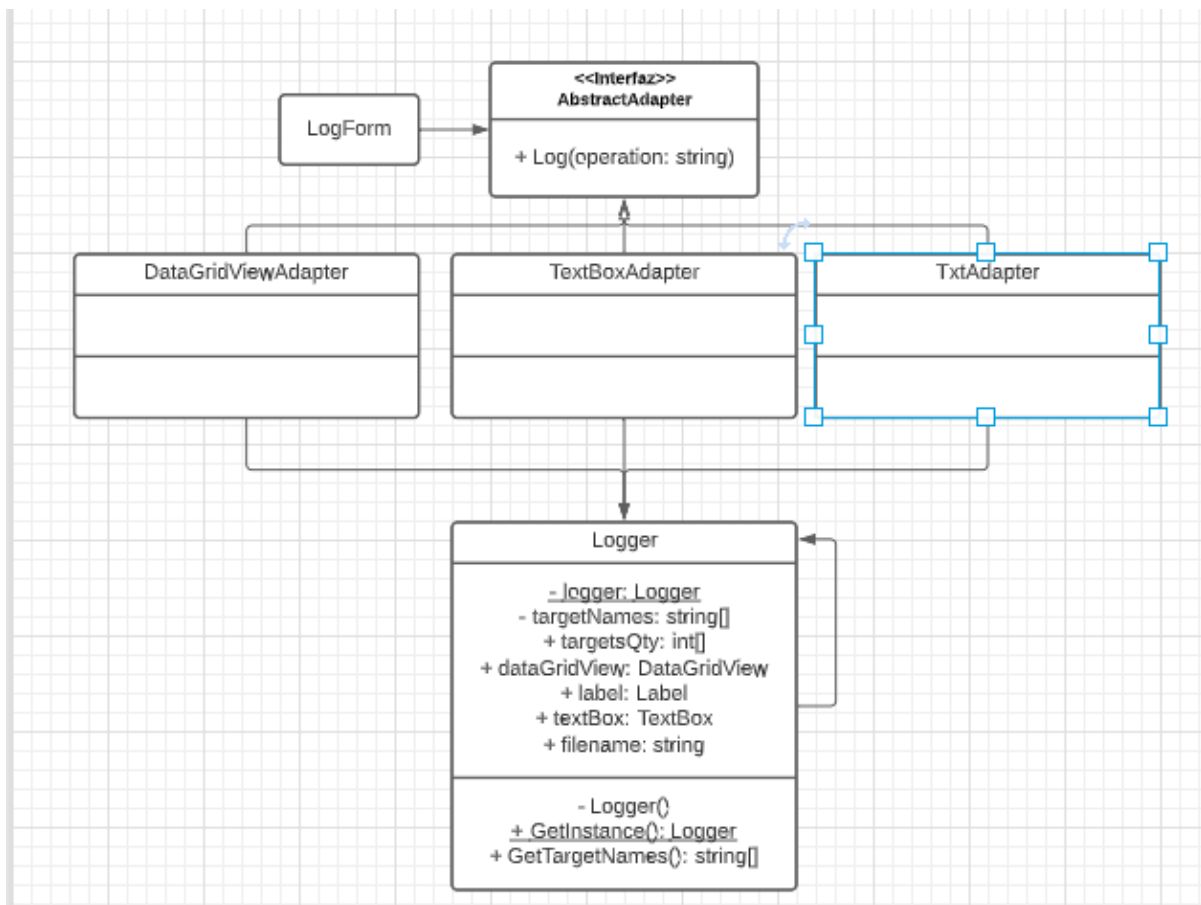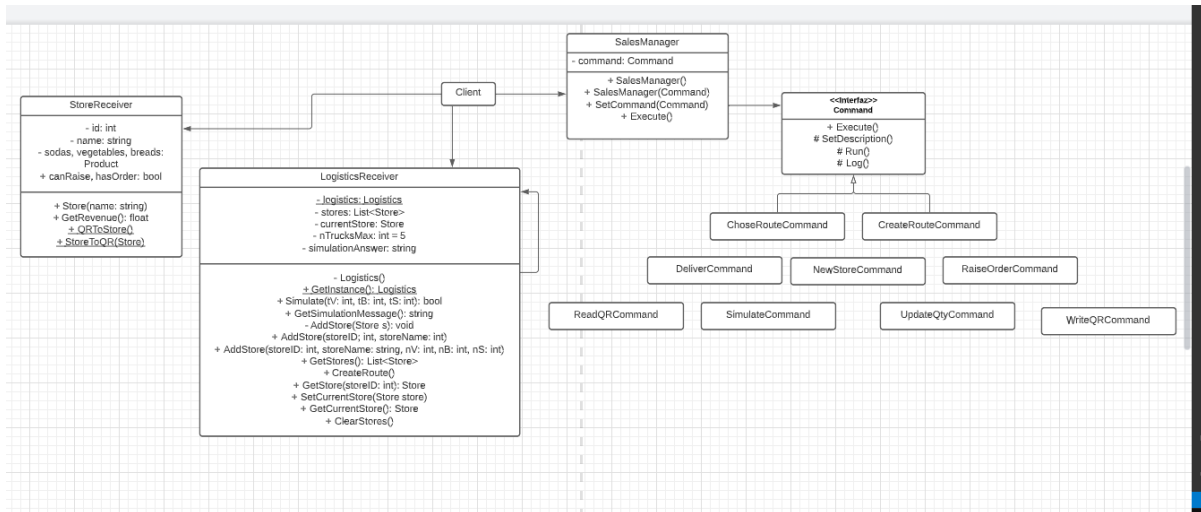
Behavioral

💡 <u>Command</u>

Command is the pattern capable to connect all the requests between senders and receivers, like disctint forms and the Logisticts class, and had the super important job of log all the operations that are made through it. This pattern also helps to decouple requests, which made the development phase of this project way easier. Lots of Commands where needed but registering the operations wasn't a problem because Command provided support for it, and it helped to have more control over the requests and allowed to parametrize objects, which helped to reuse code.

Luis