Group 12 - Design Patterns List

Factory Method:

Associated Class Names:

- GatewayPoolFactory
- ControllerPool

Why do we implement this? To encapsulate the creation process of the gateways in our system and maintain them in the gateways, rather than dealing with gateway instantiation inside the controllers.

How do we implement this? ControllerPool calls an instance of GatewayPoolFactory which creates a relevant GatewayPool depending on the implementation.

Dependency Injection:

Associated Class Names:

- Gateway Interfaces: AccountGateway, CitiesGateway, ItemsGateway, ThresholdsGateway, TradeGateway
- Use cases: Any use case class that depends on the above interface, (AccountRepository, CityManager, etc).
- Presenter Interfaces: See presenters folder (AdminWishlistPresenter, FreezingPresenter, etc).
- Controllers: Any controllers that depend on presenter interfaces.

Why do we implement this? Supporting a live trading system requires the program data to change constantly. Use Case classes need an input port interface, through which they receive data, and an output port interface, through which they can save changes performed in the system.

How do we implement this? We define a collection of gateway interfaces. Each interface has a populate method: the input port of the data. Through the populate method, data from the gateway is being sent to the use cases. In addition, the output port is the gateway save method. Each time save is called, the changes the user performs are being updated to the gateway.

Currently, there are two main implementing classes for the gateway interface. We have a “InMemory” gateway, which is used for demo mode purposes. And a json gateway, which constantly saves data to the system. The use cases do not depend on these implementations, they work the same with either one due our use of dependency injection.

Observer Pattern

Associated Class Names:

- Presenters: Any concrete class of ObservablePresenter.
- Views: Any class that ends with “View” under the “org.twelve.views” package.

Why do we implement this?

- To separate a cause (usually the underlying data of the program changing) from its effects (text/other UI components showing new data in a user friendly format).
- To prevent a violation of clean architecture where presenters would need to update UI components directly by calling their methods thus be dependent on them (outwards arrow on the clean architecture diagram).

How do we implement this?

- Presenters will fire a property changed event whenever one of their properties have changed using Java’s PropertyChangedSupport helper class (one of the replacements for the now deprecated Observable class).
- UI components inside views will listen in on property changed events and update themselves as needed.

Builder:

Associated Class Names:

- ViewBuilder
- WindowHandler

Why do we implement this? The process of building the views is complicated, and needs to be handled properly.

How do we implement this? ViewBuilder builds the controllers of the system, and builds specific scenes. The Scenes are built by iterating over a Scenes enum, and building each individual scene.