Behavioral Design Patterns

An analysis of various behavioral design patterns with examples.

Introduction

Design patterns are widely used in the Software Development industry in order to keep a project scalable and extensible but most of all help a project avoid various bugs that can be introduced by poor designs and ripple effects.

Behavioral design patterns assemble solutions for problems concerning communication and responsibilities between different objects and classes.

Chain Of Responsibility

Chain Of Responsibility is applicable on problems where an *input* must be processed by multiple functions which decide whether the input is valid, if it needs to be modified or if needs to be rejected.

Usually, this would be implemented in a single class and as requirements evolve, more logic would be added to the same class. This tactic is common when working with legacy code and can often result to impact on existing functionality, or higher software delivery times due to the added complexity.

Chain Of Responsibility provides a solution to this problem as follows:

1. Every distinct processing/filtering function is called a *Handler.*
2. Handlers implement one and only one rule and are also responsible for providing the needed functionality to set the next handler on the chain.
3. Optionally, there could be a *BaseHandler* abstract class that also implements the same interface and is responsible for implementing boilerplate functionality (such as setting the next Handler in the chain).
4. All the Handlers should implement a common interface, following the DI principle.

By following this DP it is easier to support the SRP as every handler class will only have a single reason to change: the specific filtering/processing it applies on the input data.

Furthermore, as stated above this design pattern supports the Dependency Inversion principle – by depending on the abstract *BaseHandler* and the *Handler Interface* instead of concrete implementations. This gives to the *Client* the ability to be completely unaware of the concrete implementations of each handler and thus, achieving loose coupling between the classes.

By now we have concluded to the following, comparing to a monolithic approach:

- SRP assures that our implementation will reduce the ripple effects of new changes and will enable the re-usability of our code.
- DI assures the flexibility and extensibility of our program.

The practical usage of this analysis can be found on: *Examples/chain-of-responsibility*.