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/*

Rules

The Rules design pattern has many similarities with the Chain of Responsibility. It, as well, enables developers to avoid large, unmaintainable conditional statements that can result to large Cyclomatic Dependency metrics.

Note: Cyclomatic Dependency is a metric that measures the complexity of a program. Large values of this metric indicate that a program is very complex and risky to modify.

This design pattern is particularly effective when a program needs to execute an arbitrary number of business rules that might need to change or extended in the future.

In a monolithic approach, this would start from a small if-else statement but as more business rules would be added, more complexity would be added and the class would eventually become unmaintainable (risk of changing would be high, re-usability wouldn't be possible at all).

Example:

```
WebResponseDto webResponse = WebService.GetResponse();

if (webResponse.Password.Length < 8) return Status.Retry;
else if (webResponse.Username.Contains(Regex.Try([0-9])) return Status.Invalid;
```

If more Business Rules for validating the web response are added in the future this will definitely result to the effects described above.

Instead, Rules design pattern suggests that each business rule should be implemented in a separate class and all of the rules must implement the same interface.

- Every rule should implement a common interface. For the above example this could be *IResponseValidator* that would consist of *IsMatch* and *Execute* methods.
- The *IsMatch* method could check if the rule is applicable for execution (e.g., if a rule is enabled through configuration) and the *Execute* method could execute the rule and return the *Status* based on the above example.
- Every rule should consist of a separate class. In the above example these classes could be *IsPasswordLengthOk* and *IsUsernameLettersOnly*.
- Finally, for the abstraction of the execution there can be a *RulesExecutor* that will iterate through the rules and execute the ones that are matches for the specific input / configured.
- The *Client* that needs to decide for an action based on the business rules can now simply invoke the executor.

This design pattern supports firmly the SRP and DI principles. As opposed to the monolithic conditional approach we are now able to modify every business rule individually, without affecting the execution or impacting the rest of the rules.

Also, in case more business rules are added in the future we will be able to incorporate them into our existing codebase seamlessly. Every rule is able to be unit tested individually now, adding even more value to our solution.

The practical usage of this analysis can be found on: *Examples/rules/*

Command

The Command design pattern is an implementation of the Producer-Consumer design pattern that is described in GOF. You may also find this pattern called action or transaction pattern. It is mainly used in order to decouple a class that produces requests from the class that receives and executes these requests, as well as in order to keep track of all the commands that have been executed - thus enabling the functionality of undo/redo operations. It consists of the following components:

- Invoker: Responsible for invoking the appropriate commands. The Invoker is also responsible for keeping a Buffer that holds a history of the command execution along with the state at the specific point of time – thus, supporting redo/undo operations.
- Command: There can be multiple Command classes that should implement the same ICommand interface. The concrete command classes contain all the information regarding which *Receiver* to invoke and what arguments/state to pass.
- Receiver: Provides the concrete implementation of the business logic that each command needs to be implementing.
- Client: Creates the concrete Commands and sets the respective Receiver for each one.

In some cases, there could be some variations of this design pattern. The most obvious ones are listed below:

- To reduce the number of layers, one could implement the business logic for each concrete command, inside the same class and not delegate it to the Receiver.
- Command interface could have additional methods that would be used to verify if a command can be executed.

By following this design pattern, we can effectively achieve the following things

1. Decouple the Requestor/Producer classes from the Receiver/Consumer classes. This gives the benefit of completely decoupling UI code (an example) from business logic and concrete implementations – which supports the SRP principle.
2. Every concrete Command is obvious that should do one thing only. This design pattern allows us to follow the SRP principle in an elegant way.
3. Dependency Injection can be achieved through constructors.
4. In case more commands need to be added in the future, the existing logic is never affected, thus, leading to minimized Ripple Effects and the support of the Open/Closed principle.
5. Different commands can be composite into a single command. For example, a Producer that needs to call a Restart operation might be calling multiple commands under the hood (e.g., save, clear, restart).

The practical usage of this analysis can be found on: *Examples/command/*