Facade Design Pattern

Purpose and Motivation

The Facade design pattern is a structural pattern that provides a simplified interface to a set of interfaces in the subsystem to make it easier to use. The client interacts with the facade and the facade interface is responsible for calling functions of existing subsystem.
The purpose of the facade pattern is to allow for easier access to large systems by providing a simplified interface, thus hiding complex implementation details from the client. This can be useful when building public APIs or SDKs that may be used by other developers. In order to provide a clear, simple API to access the main system you may be motivated to use the facade pattern. A facade interface object can be defined which clearly states the functions exposed by the system, while all specific implementation details of the provided function are hidden away from client callsites.

Intended Use Cases

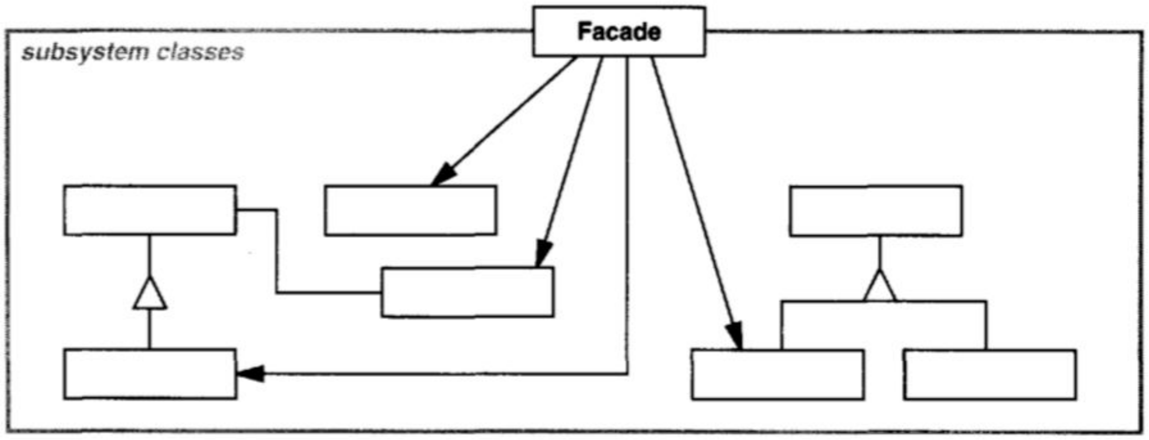
The Facade design pattern hides the structure and complexity of a subsystem from clients and provides them with clean and concise functionalities in the facade interface. More specifically, it is useful for cases where the client needs specific functions but does not need to understand the implementation details. For example, suppose there was a giant library to interface with the computer hardware. A facade design pattern can make the functionality much easier and simpler to use for a new
programmer who wishes to use the hardware to build an app without writing low-level code. This is also especially useful for simplifying a complicated API. It can also help to increase code flexibility.

Vocabulary

Client: an object or a piece of code that utilizes facade interface.
Facade: a class that serves as a front-facing interface masking more complex underlying or structural code. It wraps complicated subsystems and provides a set of functionalities to clients.
Subsystem: a collection of classes that provides some functionalities.

Structure and Runtime Behaviour

In the UML diagram, clients do not access the subsystem classes directly. Instead, the clients work through a Facade class that implements a simple interface in terms of the subsystem classes. The client depends only on the simple Facade interface and is independent of the complex subsystem.



Known Consequences

Positive Consequences

- Reduce coupling between clients and the system, minimize compilation dependencies on a subsystem, in order to provide flexibility. All of the complex operations are abstracted away from the client, so changes can be made to the subsystem without affecting the clients' behaviours.
- Hide complex details from the clients, so that programmers have the freedom to make future modifications.
- Improve usability and simplicity by providing a simple and easy entry to a subsystem for the clients to use. When a client wants to perform a complex task involving many classes in the subsystem, it only needs to interact with the façade instead of directly accessing many different objects in data structures in the subsystem.
- Provide high readability. Since façade acts as a higher-level layer, it does not usually have additional functionality, and it only collects and combines functionalities that already exists in the subsystem.

Negative Consequences

- The client is limited to the functionality provided by the facade.

- The subsystem methods are connected to the façade layer. If the structure of the subsystem changes then it will require a subsequent change to the facade layer and client methods.

NFPs

Improved NFPS

- Reusability: The Facade design pattern provides a modular and well-documented subsystem. All subclasses are classified by their functionality. In addition, the system can be used by various clients.
- Readability: The Facade design pattern provides clients and programmers with a simple and clear purpose for each method and source code. It is easy for programmers to find bugs and control flow.
- Robustness: In the facade design pattern, the facade helps subclass to check invalid inputs, defects in connected software or hardware components, and unexpected operating conditions.

Inhibited NFPS

- Complexity: The Facade design pattern provides a modular clean and easy-to-use user interface for clients rather than accessing different subclasses and learning how to use them separately.

Credits

Students:

Red (Akash Kalimili, Priyesh Patel, Alex Zandi)

Team 100% (Jian Li, Xueyao Yu, Zhixin Yuan, Xuexin Wang)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal