**NAME: KASONGO NATHAN KALONJI**

**REG\_NO: S20B23/003**

**DESIGN PATTERNS**

A collection of all-purpose fixes for recurrent design issues that software developers frequently run across are known as design patterns in the field of software development. These patterns, which are intended to enhance the quality, effectiveness, and maintainability of software systems, are often drawn from best practices and tried-and-true design concepts. These are several crucial design pattern components:

Design patterns provide many advantages, including making it easier for designers to communicate and work together, enhancing readability and quality of code, cutting down on development time and effort, and encouraging code reuse and maintainability.

Application of design patterns: A variety of computer languages and frameworks support the application of design patterns. While implementing design patterns, developers might use pre-existing frameworks and libraries or start from scratch.

Common design patterns: The Singleton pattern, Factory pattern, Adapter pattern, Observer pattern, and Strategy pattern are a few of the most widely used design patterns. Each of these patterns offers a reusable solution to a specific design issue.

Design patterns have drawbacks. Although they have many advantages, design patterns can potentially restrict flexibility when utilized inappropriately. It's critical to comprehend each pattern's context and constraints and to utilize it sparingly.

Design patterns are a useful technique for software engineers to raise the caliber and maintainability of software systems, in conclusion. Developers can produce more reliable and effective software by adhering to best practices and utilizing tried-and-true fixes for typical design issues.

**FAÇADE PATTERN**

The Facade design pattern is a structural design pattern that provides a simple interface to a complex system of objects, making it easier to use and reducing coupling between components.

It involves creating a single interface or class that provides a simplified view of a complex system of classes or subsystems. The Facade encapsulates the details of the subsystems and provides a simplified interface that clients can use to interact with the system.

It improves the usability, maintainability, and extensibility of a system by reducing coupling between components, hiding implementation details, and simplifying the interface.

Implementation: To implement the Facade pattern, you need to create a Facade class that provides a simplified interface to the complex system of classes or subsystems. The Facade class should hide the implementation details of the subsystems and provide a set of methods that clients can use to interact with the system.

Examples: The Facade pattern is commonly used in GUI toolkits, where a single class or interface provides a simplified view of the underlying system of components. It is also used in web services and APIs, where a Facade can provide a simplified interface to a complex system of backend services.

If the Facade class gets too complicated or if the subsystems change frequently, the pattern could potentially create more complexity. Also, it's crucial to watch out for the Facade class's exposure of too much information about the subsystems, as this could result in a close connection between the client and the subsystems.

In conclusion, the Facade design pattern is an effective one that offers a straightforward interface to a complicated structure of objects or subsystems. By lowering the connection between components and streamlining the user interface, it can enhance a system's usability, maintainability, and extensibility.