Design Patterns

Builder Pattern

Singleton Pattern

Observer Pattern

Factory Pattern

Strategy Pattern

Repository Pattern

Decorator Pattern

Command Pattern

Composite Pattern

By Karim Tarek

Builder Pattern:

- **Description:** Separates the construction of complex objects from their representation, allowing the same construction process to create different representations.
- **Use:** When you want to separate the construction of a complex object from its representation, allowing the same construction process to create different representations.
- **Example:** Using the ListView.builder widget, where the builder function is called for each item in the list, allowing dynamic creation of widgets based on the data.

Singleton Pattern:

- **Description:** Restricts the instantiation of a class to a single object and provides a global point of access to it.
- **Use:** When you want to ensure that only one instance of a class exists and provide a global access point to that instance.
- **Example:** Managing a global state or configuration object that needs to be accessed from multiple parts of the application, such as a settings manager or authentication provider.

Observer Pattern:

- **Description:** Defines a one-to-many dependency between objects, so that when one object changes its state, all its dependents are notified and updated automatically.
- **Use:** When you want to establish a dependency relationship between objects where changes in one object trigger updates in other objects.
- **Example:** Implementing event listeners or change listeners to respond to data changes, such as updating the UI when data is modified or notifying subscribers when a value is updated.

Factory Pattern:

- **Description:** Creates objects without exposing the instantiation logic to the client and refers to newly created objects through a common interface.
- **Use:** When you want to delegate the object creation to a factory class, decouple the client from concrete classes, or provide a flexible way to create objects.
- **Example:** Creating different types of widgets or components based on runtime conditions or user inputs, such as creating different buttons or cards based on a configuration.

Strategy Pattern:

- **Description:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime.
- **Use:** When you want to encapsulate multiple algorithms or behaviors and make them interchangeable dynamically.
- **Example:** Implementing different sorting algorithms for a list, where the sorting algorithm can be switched at runtime based on user preferences or other conditions.

Repository Pattern:

- **Description:** Mediates between the domain and data mapping layers, providing a way to retrieve and store data without exposing the underlying data storage details.
- **Use:** When you want to abstract the data access layer and provide a consistent interface for accessing and manipulating data.
- **Example:** Implementing a data repository that handles the CRUD operations for a specific data entity,

Decorator Pattern:

- **Description:** Adds additional functionality to an object dynamically without modifying its structure.
- **Use:** When you want to extend the behavior of an object at runtime or add features without altering the original class.
- **Example:** Composing widgets to modify their behavior or appearance, such as adding borders, shadows, or animations to existing widgets.

Command Pattern:

- **Description:** Encapsulates a request as an object, allowing parameterization of clients with different requests, queue or log requests, and support undoable operations.
- **Use:** When you want to decouple the sender of a request from the receiver, parameterize clients with different requests, or implement undo/redo functionality.
- **Example:** Implementing an undo/redo feature for user actions in an application, such as undoing a text input or reverting a change in a drawing app.

Composite Pattern:

- **Description:** Treats a group of objects as a single object, allowing clients to interact with individual objects and compositions uniformly.
- **Use:** When you want to represent a part-whole hierarchy of objects and treat individual objects and compositions uniformly.
- **Example:** Creating a hierarchical structure of UI elements, where a widget can contain other widgets, forming a tree-like structure for building complex user interfaces.

Karim Tarek

FLUTTER DEVELOPER