**A screenshot of a computer

Description automatically generated**

**Design Patterns**

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

**Builder**

The Builder design pattern is a creational design pattern that provides a way to construct complex objects step by step. Unlike other creational patterns that involve instantiating objects, the Builder pattern focuses on how to construct an object. This pattern is particularly useful when an object requires multiple steps to be created and when the process of creation is complex or needs to allow different representations of the object.

**Singleton**

The Singleton design pattern is one of the most widely used creational design patterns in software development. It ensures that a class has only one instance and provides a global point of access to that instance. This pattern is particularly useful when exactly one object is needed to coordinate actions across a system.

**Factory Method**

The Factory Method design pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It is used to define an interface for creating an object, but the implementation of which object to create is left to the subclasses.

**Use Cases for the Factory Method Pattern**

* **When a class cannot anticipate the class of objects it needs to create**:
  + For example, a GUI framework might need to create different types of buttons for different operating systems.

**Abstract Factory**

The Abstract Factory pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern is particularly useful when a system needs to be independent of the way its objects are created, composed, and represented.

**Prototype**

**Prototype** is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

**Adapter**

**Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.

**Bridge**

**Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

**Composite**

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

**Decorator**

**Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

**Facade**

**Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

**Flyweight**

**Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

**Proxy**

**Proxy** is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

**Chain of Responsibility.**

The Chain of Responsibility pattern is a behavioral design pattern that allows an object to pass a request along a chain of handlers. When a request is made, it is passed through this chain until a handler processes it. Each handler in the chain decides either to process the request or pass it along to the next handler in the chain.

**Command.**

The Command pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of requests. It also enables the support of undoable operations.

**Iterator.**

The Iterator pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It allows you to traverse elements of a collection without needing to know the specific implementation of the collection.

**Mediator.**

The Mediator pattern is a behavioral design pattern that promotes loose coupling between objects by encapsulating how these objects interact with each other. It centralizes communication logic between objects, reducing dependencies between them. Instead of objects communicating directly, they communicate through a mediator object, which handles the interactions.

**Memento.**

**Memento** is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

**Observer.**

**Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

**State.**

The State design pattern is a behavioral pattern that allows an object to alter its behavior when its internal state changes. This pattern is particularly useful when an object's behavior depends on its state and must change dynamically based on that state.

**Strategy.**

The Strategy design pattern is a behavioral design pattern that enables an algorithm's behavior to be selected at runtime. It defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family.

**Template Method.**

The Template Method design pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of the algorithm without changing its structure.

**Visitor**

The Visitor design pattern is a behavioral design pattern that allows adding new behaviors to existing classes without changing their structure. It is useful when you have a set of classes with different interfaces and you want to perform operations on these classes that are not supported by their interfaces.