**Creational Patterns**

These patterns deal with **object creation mechanisms, trying to create objects in a manner suitable to the situation.**

1. **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Builder**: Separates the construction of a complex object from its representation so that the same construction process can create different representations.
3. **Factory Method**: Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.
4. **Prototype**: Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
5. **Singleton**: Ensures a class has only one instance and provides a global point of access to it.

**Structural Patterns**

These patterns deal with **object composition, creating relationships between objects to form larger structures.**

1. **Adapter**: Allows objects with incompatible interfaces to work together by converting the interface of a class into another interface expected by the clients.
2. **Bridge**: Separates an abstraction from its implementation so that the two can vary independently.
3. **Composite**: Composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.
4. **Decorator**: Adds new functionality to an object dynamically without altering its structure.
5. **Facade**: Provides a simplified interface to a complex subsystem of classes, making it easier to use.
6. **Flyweight**: Reduces the cost of creating and manipulating a large number of similar objects by sharing common parts.
7. **Proxy**: Provides a surrogate or placeholder for another object to control access to it.

**Behavioral Patterns**

These patterns focus on communication between objects, what goes on between objects and how responsibilities are distributed.

1. **Chain of Responsibility**: Passes a request along a chain of handlers, where each handler can either process the request or pass it along to the next handler in the chain.
2. **Command**: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
3. **Interpreter**: Defines a grammar for a language and provides an interpreter to interpret sentences in the language.
4. **Iterator**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
5. **Mediator**: Defines an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from referring to each other explicitly.
6. **Memento**: Allows capturing and externalizing an object’s internal state without violating encapsulation, so the object can be restored to this state later.
7. **Observer**: Defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
8. **State**: Allows an object to alter its behavior when its internal state changes, appearing as if the object changed its class.
9. **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The algorithm can vary independently from clients that use it.
10. **Template Method**: Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
11. **Visitor**: Defines a new operation to a set of objects without changing the classes of the elements on which it operates.

These patterns are widely used in software engineering to build robust, scalable, and maintainable systems. They provide a common language for developers and help in solving recurring design problems.