Creational Patterns

* Creational patterns provide techniques for creating objects and managing the life cycle of the objects.
  + Avoid new Operator

|  |  |
| --- | --- |
| **Pattern Name** | **Pattern Description** |
| Abstract Factory | Provides an interface for creating families of related or dependent objects without specifying their concrete classes. |
| Factory Method | Deals with the problem of creating related objects without specifying the exact class of object that will be created. |
| Singleton | This pattern ensures a class has only one instance and provides a global(app-level) point of access to it. |
| Prototype | Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. |
| Builder | Separates the construction of a complex object from its representation, thus enabling the same construction process to create various representations. |

Structural Patterns

* Structural design pattern describes how classes and objects can be combined to form larger structures using object composition.

|  |  |
| --- | --- |
| **Pattern Name** | **Pattern Description** |
| Adapter | This pattern lets classes work together that could not otherwise because of incompatible interfaces. |
| Bridge | This pattern decouples an abstraction from its implementation so that they become loosely coupled. |
| Composite | This pattern allows aggregating objects such that individual objects and composition of objects can be treated uniformly. |
| Decorator | This pattern attaches additional responsibilities to an object dynamically while keeping the interface same |
| Façade | This pattern provides a simpler interface to a larger and more complex system such as a class library or a complex API. |
| Flyweight | This pattern minimizes memory usage by sharing common data between objects. |
| Proxy | Proxy is a surrogate or placeholder class for another class mostly done with an intention of intercepting access to the said class. |

Behavioral Patterns

* Behavioral patterns provide guidelines for enabling communication between two objects.

|  |  |
| --- | --- |
| **Pattern Name** | **Pattern Description** |
| Chain of Responsibility | This pattern defines a chain of processing objects in a chain in such a way that the incoming request is processed by each processing objects in sequence. |
| Command | In this pattern an object is used to represent and encapsulate all the information needed to call a method at a later time. |
| Interpreter | This pattern defines a representation for a given language’s grammar along with an interpreter that uses the representation to interpret sentences in the language. |
| Iterator | This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |
| Mediator | In this pattern communication between objects is encapsulated with a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. |
| Memento | Mementos capture and externalize an object’s internal state allowing the object to be restored to this state later. |
| Observer | An observable object called ‘Subject’ maintains a list of objects called ‘Observers’. Subject notifies the observers of any state changes. |
| State | State pattern allows an object to alter its behavior when its internal state changes. |
| Strategy | This pattern defines a family of algorithms, encapsulate each one, and make them interchangeable. |
| Template Method | Pattern defines steps of an algorithm as methods in an interface while allowing subclasses to override some steps in the overall algorithm. |
| Visitor | Pattern separates an algorithm from the object structure on which it operates, which provides the ability to add new operations to existing object structures without modifying those structures. |