designPatterns

	Desc	Applicability	Pros and Cons	Repations with other patterns
Factory method	`Factory Method` is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. - create object without exposing the creation logic to the client and refer to newly created object using a common interface.	- Use the `Factory Method` when you don't know beforehand the exact types and dependencies of the objects your code should work with. - Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components. - Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.	✓ You avoid tight coupling between the creator and the concrete products. ✓ Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support. ✓ Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code. ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.	- Many designs start by using `Factory Method` (less complicated and more customizable via subclasses) and evolve toward `Abstract Factory`, `Prototype`, or ` Builder` (more flexible, but more complicated). - `Abstract Factory` classes are often based on a set of `Factory Methods`, but you can also use `Prototype` to compose the methods on these classes. - You can use `Factory Method` along with ` Iterator` to let collection subclasses return different types of iterators that are compatible with the collections. - Prototype isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, Prototype requires a complicated initialization of the cloned object. Factory Method is based on inheritance but doesn't require an initialization step. - Factory Method is a specialization of Template Method. At the same time, a Factory Method may serve as a step in a large Template Method.
Abstract Factory	`Abstract Factory` is a creational design pattern that lets you produce families of related objects without specifying their concrete classes. - a super-factory which creates other factories, also called as `factory of factories`.	- Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products—they might be unknown beforehand or you simply want to allow for future extensibility.	✓ You can be sure that the products you're getting from a factory are compatible with each other. ✓ You avoid tight coupling between concrete products and client code. ✓ Single Responsibility Principle. You can extract the product creation code into one place, making the code easier to support. ✓ Open/Closed Principle. You can introduce new variants of products without breaking existing client code. ✗ The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.	- Many designs start by using `Factory Method` (less complicated and more customizable via subclasses) and evolve toward `Abstract Factory`, `Prototype`, or ` Builder` (more flexible, but more complicated). - Builder focuses on constructing complex objects step by step. Abstract Factory specializes in creating families of related objects. Abstract Factory returns the product immediately, whereas Builder lets you run some additional construction steps before fetching the product. - Abstract Factory classes are often based on a set of Factory Methods, but you can also use Prototype to compose the methods on these classes. - Abstract Factory can serve as an alternative to Facade when you only want to hide the way the subsystem objects are created from the client code. - You can use Abstract Factory along with Bridge. This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code. Abstract Factories, Builders and Prototypes can all be implemented as Singletons.
Builder	- builder pattern builds a complex object sung simgle objects and using a step-by-step approach. - a Builder class builds the final object step by step, this builder is independent of other objects. - Unlike other creational patterns, Builder doesn't require products to have a common interface. That makes it possible to produce different products using the same construction process.	- Use the Builder pattern to get rid of a "telescopic constructor". - Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses). - Use the Builder to construct Composite trees or other complex objects.	✓ You can contruct objects step by step, defer contruction steps or run steps recursively. ✓ You can reuse the same contruction code when building various prepresentations of products. ✓ `Single responsibility Principle`. You can isolated complex contruction code from the bisiness logic of the product. ✗ The overall complexity of code increases since the pattern requires creating multiple new classes.	- Many designs start by using `Factory Method` (less complicated and more customizable via subclasses) and evolve toward `Abstract Factory`, `Prototype`, or ` Builder` (more flexible, but more complicated). - `Builder` focuses on constructing complex objects step by step. `Abstract Factory` specializes in creating families of related objects. `Abstract Factory` returns the product immediately, whereas ` Builder` lets you run some additional construction steps before fetching the product. - You can use `Builder` when creating complex `Composite` trees because you can program its construction steps to work recursively. - You can combine `Builder` with `Bridge`: the director class plays the role of the abstraction, while different builders act as implementations. - `Abstract Factories`, `Builders` and ` Prototypes` can all be implemented as ` Singletons`.
Prototype	`Prototype` is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.	- Use the `Prototype` pattern when your code shouldn't depend on the concrete classes of objects that you need to copy. - Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.	✓ You can clone objects without coupling to their concrete classes. ✓ You can get rid of repeated initialization code in favor of cloning pre-built prototypes. ✓ You can produce complex objects more conveniently. ✓ You get an alternative to inheritance when dealing with configuration presets for complex objects. ✗ Cloning complex objects that have circular references might be very tricky.	- Many designs start by using Factory Method (less complicated and more customizable via subclasses) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, but more complicated). - `Abstract Factory` classes are often based on a set of `Factory Methods`, but you can also use `Prototype` to compose the methods on these classes. - `Prototype` can help when you need to save copies of Commands into history. - Designs that make heavy use of ` Composite` and `Decorator` can often benefit from using `Prototype`. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch. - Sometimes Prototype can be a simpler alternative to Memento. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish. - Abstract Factories, Builders and Prototypes can all be implemented as Singletons.
Singleton	`Singleton` is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.	- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program. - Use the Singleton pattern when you need stricter control over global variables.	✓ You can be sure that a class has only a single instance. ✓ You gain a global access point to that instance. ✓ The singleton object is initialized only when it's requested for the first time. ✗ Violates the Single Responsibility Principle. The pattern solves two problems at the time. ✗ Violates the Single Responsibility Principle. The pattern solves two problems at the time. ✗ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times. ✗ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.	- A Facade class can often be transformed into a Singleton since a single facade object is sufficient in most cases. - `Flyweight` would resemble `Singleton` if you somehow managed to reduce all shared states of the objects to just one flyweight object. - `Abstract Factories`, `Builders` and ` Prototypes` can all be implemented as Singletons.
Chain of Responsibility	`Chain of Responsibility` is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.	- Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand. - Use the pattern when it's essential to execute several handlers in a particular order. - Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.	✓ You can control the order of request handling. ✓ Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations. ✓ Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code. ✗ Some requests may end up unhandled.	- `Chain of Responsibility, Command, Mediator and Observer` address various ways of connecting senders and receivers of requests: + `Chain of Responsibility` passes a request sequentially along a dynamic chain of potential receivers until one of them handles it. + `Command` establishes unidirectional connections between senders and receivers. + `Mediator` eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object. + `Observer` lets receivers dynamically subscribe to and unsubscribe from receiving requests. - `Chain of Responsibility` is often used in conjunction with `Composite`. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree. - Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request. - Chain of Responsibility and Decorator have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.
Command	Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations. Command is behavioral design pattern that converts requests or simple operations into objects.	- Use the Command pattern when you want to parametrize objects with operations. - Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely. - Use the Command pattern when you want to implement reversible operations.	✓ Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations. ✓ Open/Closed Principle. You can introduce new commands into the app without breaking existing client code. ✓ You can implement undo/redo. ✓ You can implement deferred execution of operations. ✓ You can assemble a set of simple commands into a complex one. ✗ The code may become more complicated since you're introducing a whole new layer between senders and receivers.	- `Chain of Responsibility, Command, Mediator and Observer` address various ways of connecting senders and receivers of requests: + `Chain of Responsibility` passes a request sequentially along a dynamic chain of potential receivers until one of them handles it. + `Command` establishes unidirectional connections between senders and receivers. + `Mediator` eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object. + `Observer` lets receivers dynamically subscribe to and unsubscribe from receiving requests. - Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request. - You can use Command and Memento together when implementing "undo". In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed. - Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents. - Prototype can help when you need to save copies of Commands into history. - You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes.
Interator	Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). Thanks to the Iterator, clients can go over elements of different collections in a similar fashion using a single iterator interface.	- Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons). - Use the pattern to reduce duplication of the traversal code across your app. - Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.	✓ Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes. ✓ Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything. ✓ You can iterate over the same collection in parallel because each iterator object contains its own iteration state. ✓ For the same reason, you can delay an iteration and continue it when needed. ✗ Applying the pattern can be an overkill if your app only works with simple collections. ✗ Using an iterator may be less efficient than going through elements of some specialized collections directly.	- You can use Iterators to traverse Composite trees. - You can use Factory Method along with Iterator to let collection subclasses return different types of iterators that are compatible with the collections. - You can use Memento along with Iterator to capture the current iteration state and roll it back if necessary. - You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.
Mediator	Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object. Mediator is a behavioral design pattern that reduces coupling between components of a program by making them communicate indirectly, through a special mediator object.	- Use the Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes. The pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component from the rest of the components. - Use the pattern when you can't reuse a component in a different program because it's too dependent on other components. - Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.	✓ Single Responsibility Principle. You can extract the communications between various components into a single place, making it easier to comprehend and maintain. ✓ Open/Closed Principle. You can introduce new mediators without having to change the actual components. ✓ You can reduce coupling between various components of a program. ✓ You can reuse individual components more easily. ✓ Over time a mediator can evolve into a God Object.	- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests: Chain of Responsibility passes a request sequentially along a dynamic chain of potential receivers until one of them handles it. Command establishes unidirectional connections between senders and receivers. Mediator eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object. Observer lets receivers dynamically subscribe to and unsubscribe from receiving requests. Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes. Facade defines a simplified interface to a subsystem of objects, but it doesn't introduce any new functionality. The subsystem itself is unaware of the facade. Objects within the subsystem can communicate directly. Mediator centralizes communication between components of the system. The components only know about the mediator object and don't communicate directly. The difference between Mediator and Observer is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.
Memetor	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. Memento is a behavioral design pattern that allows making snapshots of an object's state and restoring it in future.	- Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object. - Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.	✓ You can produce snapshots of the object's state without violating its encapsulation. ✓ You can simplify the originator's code by letting the caretaker maintain the history of the originator's state. ✗ The app might consume lots of RAM if clients create mementos too often. ✗ Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos. ✗ Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays untouched.	You can use Command and Memento together when implementing "undo". In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed. You can use Memento along with Iterator to capture the current iteration state and roll it back if necessary. Sometimes Prototype can be a simpler alternative to Memento. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.
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. Observer is a behavioral design pattern that allows some objects to notify other objects about changes in their state.	- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically. You can often experience this problem when working with classes of the graphical user interface. For example, you created custom button classes, and you want to let the clients hook some custom code to your buttons so that it fires whenever a user presses a button. The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects. You can add the subscription mechanism to your buttons, letting the clients hook up their custom code via custom subscriber classes. Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases. The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.	✓ Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface). ✓ You can establish relations between objects at runtime. ✗ Subscribers are notified in random order.	Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests: Chain of Responsibility passes a request sequentially along a dynamic chain of potential receivers until one of them handles it. Command establishes unidirectional connections between senders and receivers. Mediator eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object. Observer lets receivers dynamically subscribe to and unsubscribe from receiving requests. The difference between Mediator and Observer is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that....
State	State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. State design pattern is a behavioral design pattern that is based on Finite State Machine.	Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently. Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields. Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.	✓Simplify the code of the context by eliminating bulky state machine conditionals. ✗ Applying the pattern can be overkill if a state machine has only a few states or rarely changes.	Bridge, State, Strategy (and to some degree Adapter) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.
Strategy	Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. Strategy is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable inside original context object.	Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime. Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior. Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic. Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.	✓You can swap algorithms used inside an object at runtime. ✓You can isolate the implementation details of an algorithm from the code that uses it. ✓ You can replace inheritance with composition. ✗ If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern. ✗ Clients must be aware of the differences between strategies to be able to select a proper one. ✗ A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.	Bridge, State, Strategy (and to some degree Adapter) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves. Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents.... Decorator lets you change the skin of an object, while Strategy lets you change the guts. Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it's static. Strategy works on the object level, letting you switch behaviors at runtime. State can be considered as an extension of Strategy. Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects. Strategy makes these objects completely independent and unaware of each other. However, State doesn't restrict dependencies between concrete states, letting them alter the state of the context at will.
Template Method	Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure. Template Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.	Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure. Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.	✓ You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm. ✓You can pull the duplicate code into a superclass. ✗ Some clients may be limited by the provided skeleton of an algorithm. ✗ You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass. ✗ Template methods tend to be harder to maintain the more steps they have.	Factory Method is a specialization of Template Method. At the same time, a Factory Method may serve as a step in a large Template Method. Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it's static. Strategy works on the object level, letting you switch behaviors at runtime.
Visitor	Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate. Visitor is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.	Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree). Use the Visitor to clean up the business logic of auxiliary behaviors. Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.	✓ A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure. ✗ You need to update all visitors each time a class gets added to or removed from the element hierarchy. ✗ Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.	You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes. You can use Visitor to execute an operation over an entire Composite tree. You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.
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. Using the Composite pattern makes sense only when the core model of your app can be represented as a tree. Composite is a structural design pattern that allows composing objects into a tree-like structure and work with the it as if it was a singular object.	Use the Composite pattern when you have to implement a tree-like object structure. Use the pattern when you want the client code to treat both simple and complex elements uniformly.	✗ It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.	You can use Builder when creating complex Composite trees because you can program its construction steps to work recursively. Chain of Responsibility is often used in conjunction with Composite. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree. You can use Iterators to traverse Composite trees. You can use Visitor to execute an operation over an entire Composite tree. Composite and Decorator have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects. Designs that make heavy use of Composite and Decorator can often benefit from using Prototype. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.