Three groups of design patterns

- Creational
- Structural
- Behavioral

Design pattern

Well tested solution to a common problem in a context. They are best practices on how classes and objects might be arranged to accomplish a result. Patterns enumerate consequences for evaluating design alternatives, costs, benefits.

Four elements of Design patterns

1. A name that uniquely identifies the pattern.
2. A problem description that describes the situation in which the pattern can be used.
3. A solution stated as a set of collaborating classes and interfaces.
4. A set of consequences that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

Design Pattern classes

- the client class accesses the pattern classes.
- The pattern interface is the part of the pattern that is visible to the client class (might be an interface or abstract class).
- The implementer class provides low level behavior of the pattern, usually more than one.
- The extender class specialized an implementor class to provide different implementations of the pattern. Usually represented by the developer.

Creational Patterns

Deals with creation of objects

- Abstract Factory
- Builder
- Factory
- Prototype
- Singleton

1. Abstract Factory
   - Allows you to create families of related objects without specifying a concrete class
   - Use when you have many object that can be added or changed dynamically during runtime
2. Builder
   - Allows you to create objects made from other objects
   - When you want the creation of these parts to be independent of the main object
3. Factory Method
   - Allows you to create objects without specifying the exact class of objects that will be created at runtime.
   - When a method returns one of several possible classes that share a common superclass.
4. Prototype
   - Allows you to create new objects by cloning(copying) other objects.
5. Singleton
   - Pattern used when you want to instantiate only one object of a class.
   - I uses lazy instantiation which means if the object isn't needed it isn't created.

Structural

Deals with how classes are designed. How things like inheritance, composition and aggregation can be used to provide extra functionality.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

1. Adapter
   - Allows 2 incompatible interfaces to work together.
   - Used when the client expects a (target) interface
   - It is often used to make existing classes work together with others without modifying their source code.
2. Bridge
   - Problem: to decouple an abstraction from its implementation so the two can vary independently. The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.
   - Solution: Abstraction is visible to the client. Abstraction maintains a reference to its corresponding implementor instance.
   - When to use it? When you want to be able to change both the abstractions (abstract classes) and concrete classes independently
3. Composite
   - Allows you to treat a group of objects the same way as a single instance of the same type of object.
   - Lets clients treat individual objects and compositions uniformly.
   - They allow you to represent part-whole hierarchies
   - You can structure data, or represent the inner workings of every part of a whole object individual
4. Decorator
   - The Decorator allows you to add behavior dynamically without affecting the behavior of other objects from the same class.
   - You could use it when you want the capabilities of inheritance with subclasses, but you need to add functionality at runtime
   - It is more flexible than inheritance
5. Facade
   - Allows you to create a simplified interface that performs many other actions behind the scenes
   - Can I withdraw $50 from the bank?
   - Check if the checking account is valid
   - Check if the security code is valid
   - Check if funds are available
   - Make changes accordingly
6. Flyweight
   - Used when you need to create a large number of similar objects (> 100K)
   - To reduce memory usage, you share Objects that are similar in some way rather than creating new ones
   - Intrinsic State: Color
   - Extrinsic State: Size
7. Proxy
   - Provide a class which will limit access to another class
   - You may do this for security reasons, because an Object is intensive to create, or is accessed from a remote location.

Behavioral

Specifically concerned with communication between objects as the program is running:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

1. Chain of Responsibility
   - This pattern sends data to an object and if that object can't use it, it sends it to any number of other objects that may be able to use it
2. Command
   - Pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time.
   - This information includes the method name, the object that owns the method and values for the method parameters
   - Allows you to store lists of code that is executed at a later time or many times.
3. Interpreter
   - Useful when used with Java Reflection
   - It is used to convert one representation of data into another
4. Iterator
   - The Iterator pattern provides you with a uniform way to access different collections of objects
   - If you get an Array, ArrayList and Hashtable of Objects, you pop out an iterator for each and treat them the same
   - This provides a uniform way to cycle through different collections
5. Mediator
   - Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
6. Memento
   - A way to store previous states of an object easily
   - Memento: The basic object that is stored in different states
   - Originator: Sets and Gets values from the currently targeted Memento. Creates new Mementos and assigns current values to them
7. Observer
   - When you need other objects to receive an update when another object changes.
   - The Subject maintains a list of Observers and notifies them automatically when its internal state changes usually by calling one of their methods.
   - Loose coupling is a benefit. The Subject (publisher) doesn't need to know anything about the Observers(subscribers)
8. State
   - Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
   - Context (Account): Maintains an instance of a ConcreteState subclass that defines the current state
   - State: Defines an interface for encapsulating the behavior associated with a particular state of the Context
   - Concrete State: Each subclass implements a behavior associated with a state of Context
9. Strategy
   - When you want to define a class that will have one behavior that is similar to other behaviors in a list.
   - When you need to use one of several behaviors dynamically
   - Instead of implementing a single algorithm directly code receives run-time instructions as which in a family of algorithms to use.
10. Template
    - Used to create a group of subclasses that have to execute a similar group of methods
    - You create an abstract class that contains a method called the Template Method
    - The Template Method contains a series of method calls that every subclass object will call
    - The subclass objects can override some of the method calls
    - Defines the program skeleton of of an algorithm of an operation, deferring some steps to subclasses.
11. Visitor
    - Allows you to add methods to classes of different types without much altering those classes
    - You can make completely different methods depending on the class used