Design Patterns in Python

Learn how to leverage design patterns in Python applications

Ramesh S

What is a Design Pattern?

- A Design Pattern describes a problem and a general approach to solving it.
- Design patterns are used to represent the pattern used by developers to create software or web application.
- These patterns are selected based on the requirement analysis.
- Programs can be made easily understandable and extensible by using Design Patterns.

Structure of a Design Pattern

Pattern Name	Describes pattern in short & effective manner
Intent/Motive	Describes what pattern does
Applicability	Describes situations where pattern is applicable
Participants and Consequences	Includes classes and objects that are part of the pattern with a list of consequences that exist

What Constitutes a Design Pattern in Python?

- Pattern Name
- Intent
- Aliases
- Motivation
- Problem
- Solution
- Structure
- Participants
- Constraints
- Sample Code

Advantages of Design Patterns

- Patterns provide developer a selection of tried and tested solutions for the specified problems.
- All design patterns are language neutral.
- Patterns help to achieve communication and maintain well documentation.
- It includes a record of accomplishment to reduce any technical risk to the project.
- Design patterns are highly flexible to use and easy to understand.

Types of Design Patterns

Creational Patterns

Concerned with creating objects

Eg.: Factory, Builder

Structural Patterns

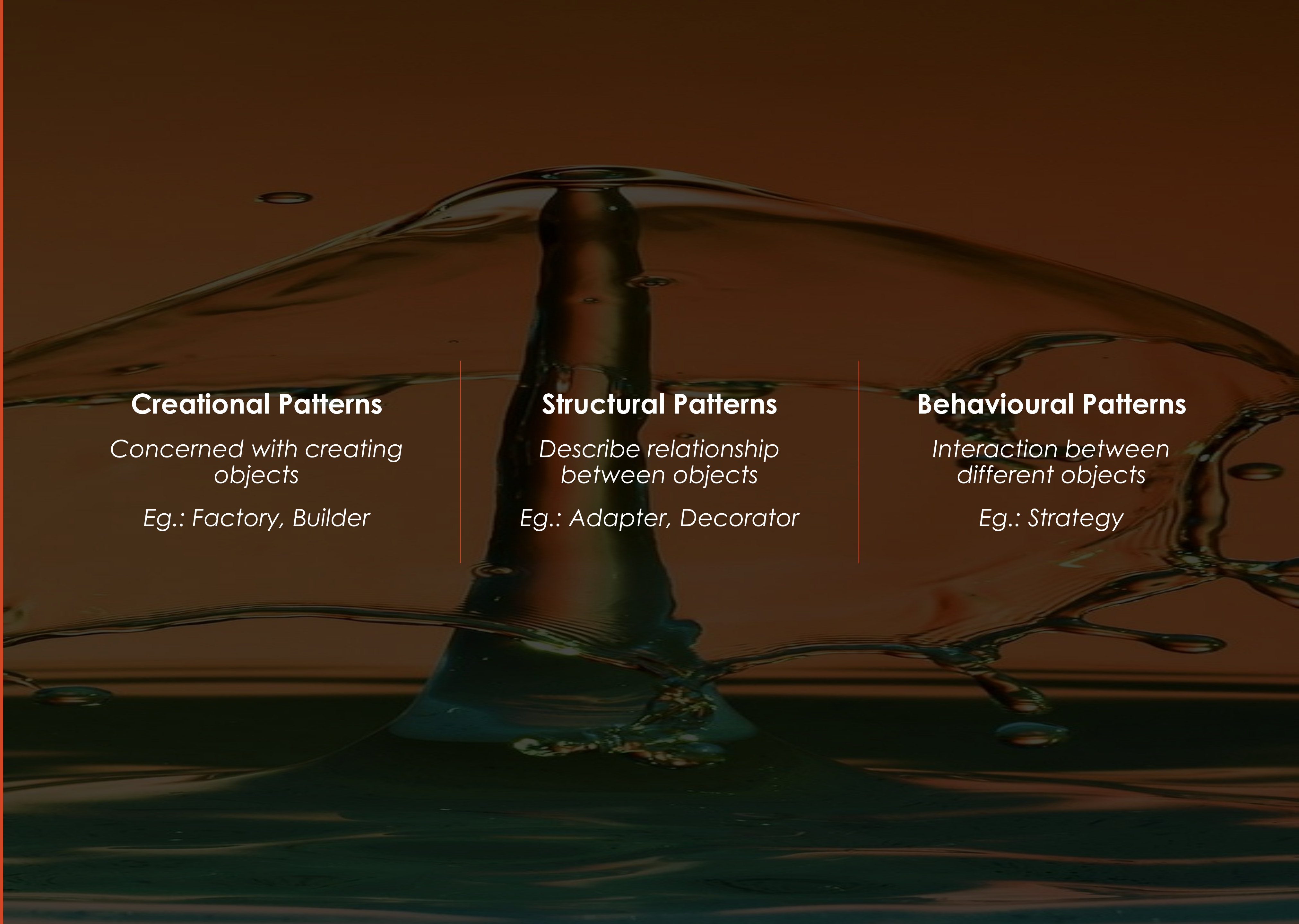
Describe relationship between objects

Eg.: Adapter, Decorator

Behavioural Patterns

Interaction between different objects

Eg.: Strategy

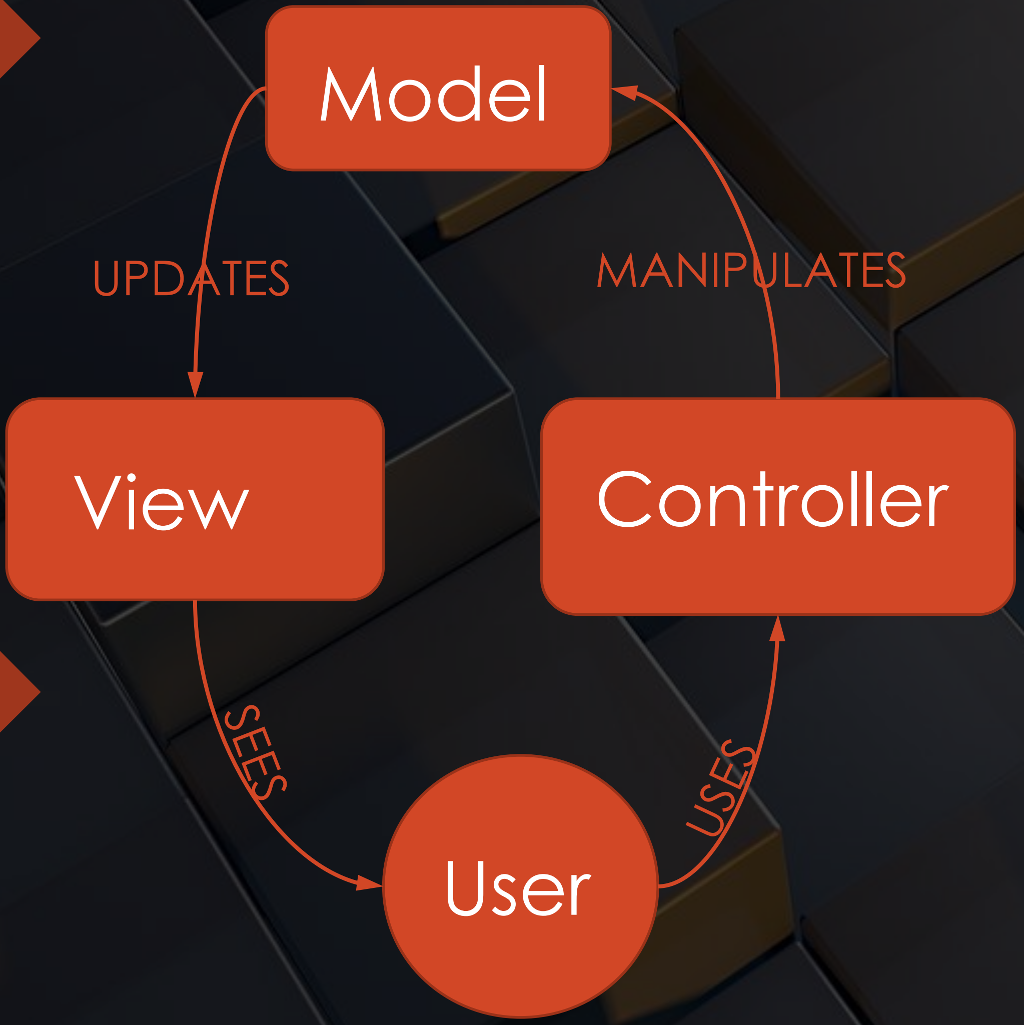


Design Patterns

- Model View Controller Pattern
- Singleton pattern
- Factory pattern
- Builder Pattern
- Prototype Pattern
- Facade Pattern
- Command Pattern
- Adapter Pattern
- Prototype Pattern
- Decorator Pattern

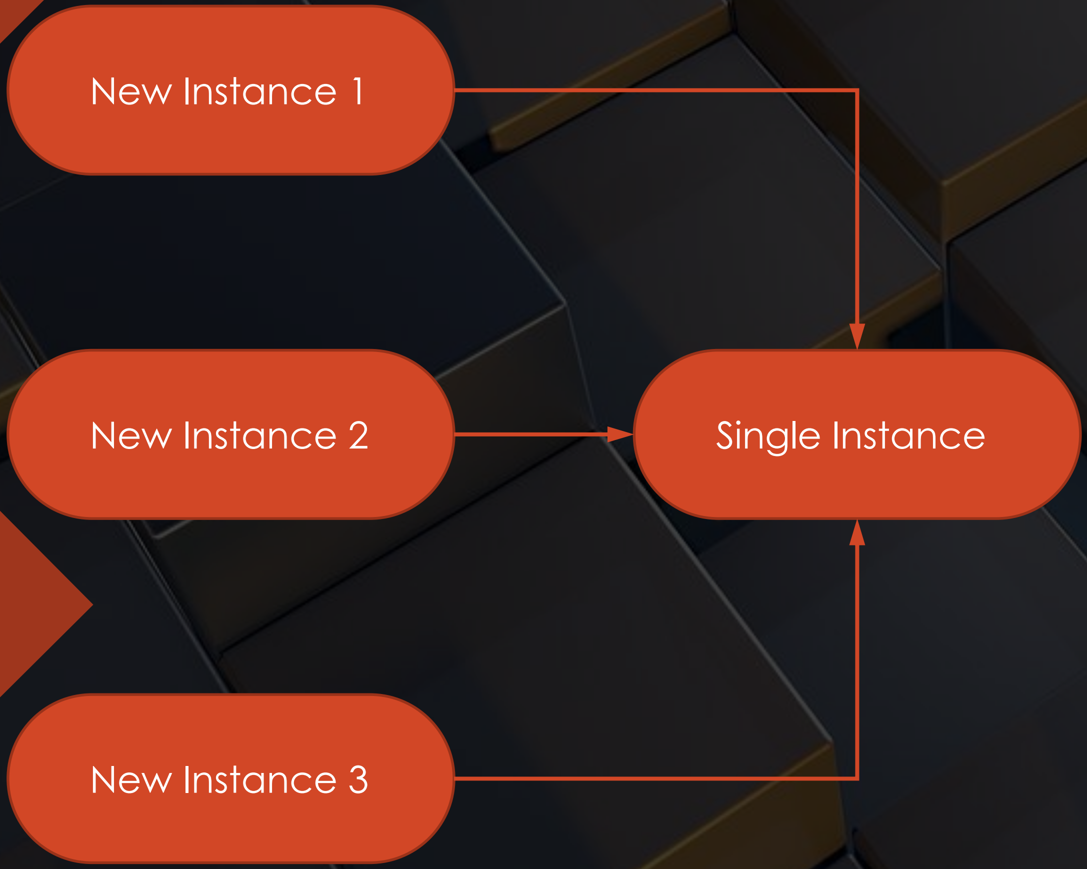
- Proxy Pattern
- Chain of Responsibility Pattern
- Observer Pattern
- State Pattern
- Strategy Pattern
- Template Pattern
- Flyweight Pattern
- Abstract Factory Pattern
- Object Oriented Pattern

Model View Controller Pattern



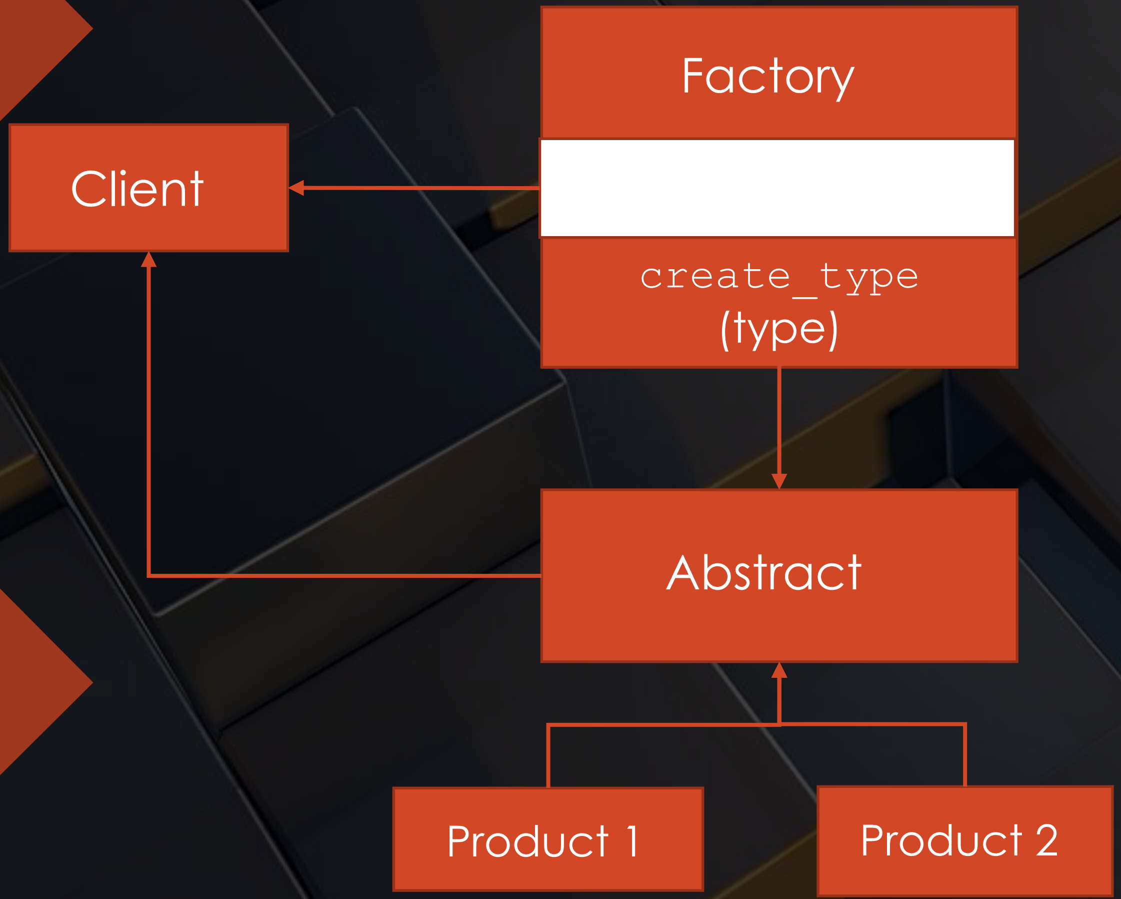
- Model View Controller is the most commonly used design pattern.
- Developers find it easy to implement this design pattern.
- *Model*:
  - Pure application logic, which interacts with the database
  - Includes all data to represent data to end user
- *View*:
  - Represents HTML files, that interact with end user
- *Controller*:
  - Intermediary between View and Model
  - Listens to events triggered by View queries Model for the same

Singleton Pattern

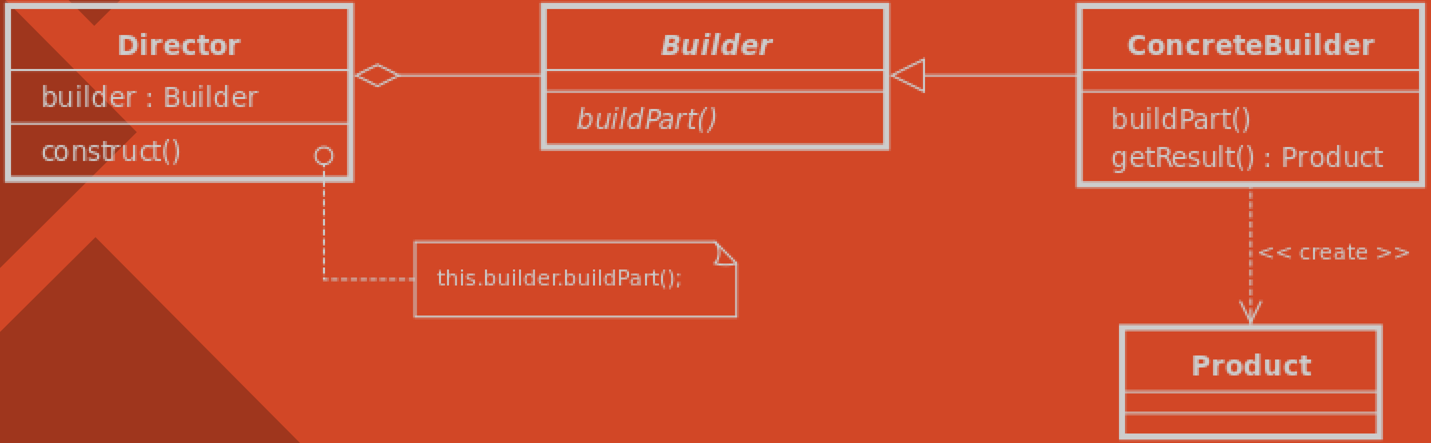


- This pattern restricts the instantiation of a class to one object.
- It is a type of creational pattern and involves only one class to create methods and specified objects.
- It provides a global point of access to the instance created.

Factory Pattern



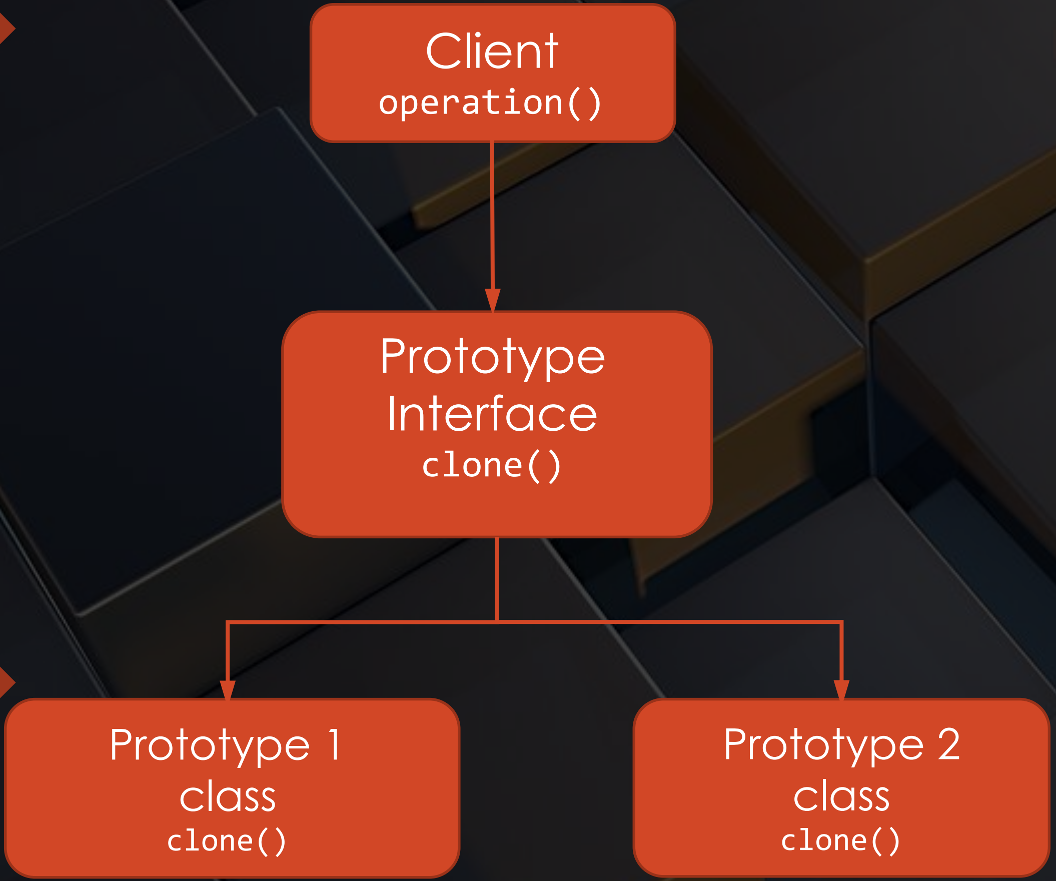
- Objects are created without exposing the logic to client and referring to the newly created object using a common interface.
- Factory patterns are implemented in Python using factory method.
- When a user calls a method such that it *takes a string and returns a new object* is implemented through factory method.
- The type of object used in factory method is determined by string which is passed through method.



Builder Pattern

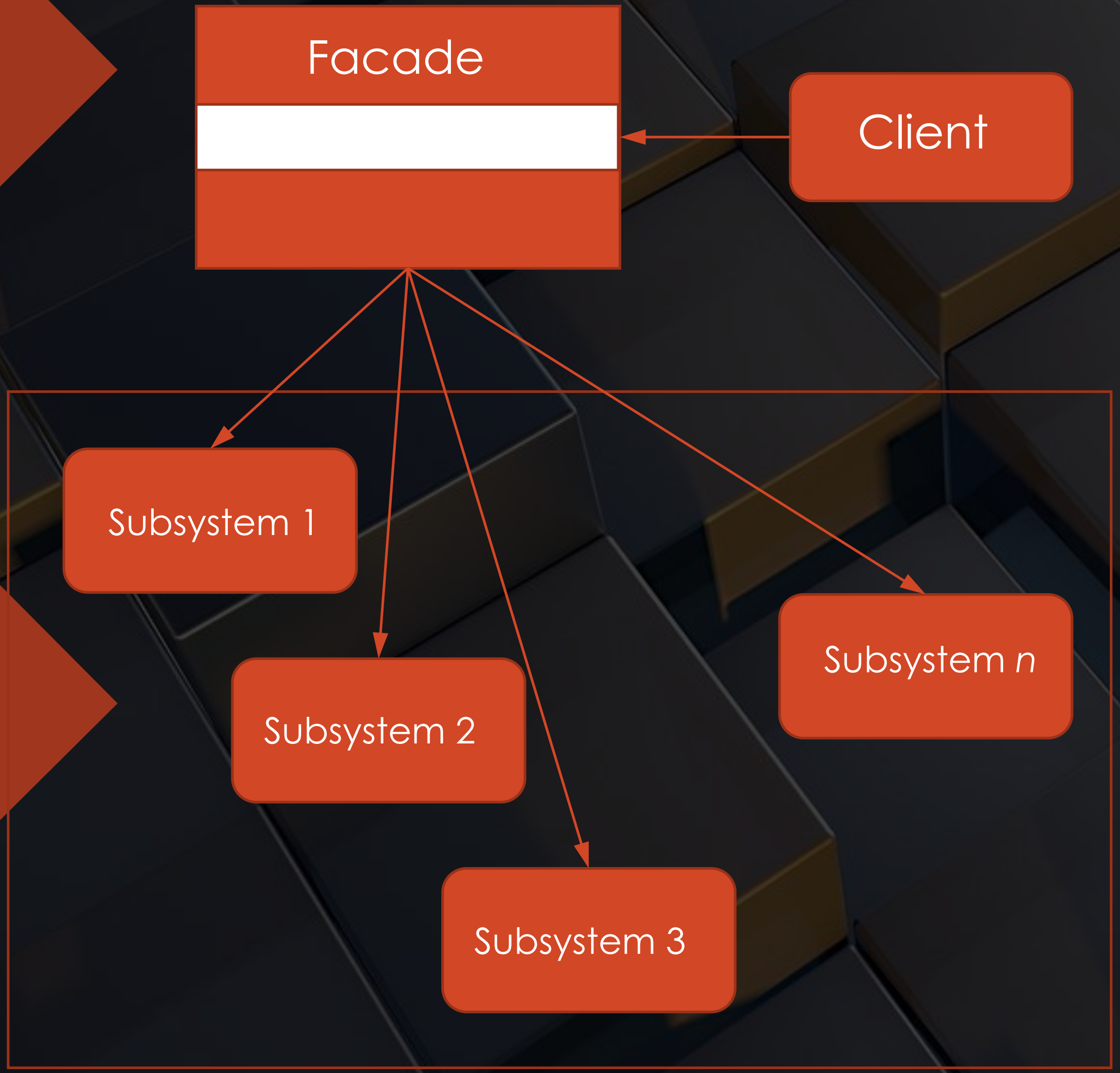
- Builder Pattern is a unique design pattern which helps in building complex object using simple objects and uses an algorithmic approach.
- In this design pattern, a builder class builds the final object in step-by-step procedure.
- This builder is independent of other objects.
- Advantages of Builder Pattern
  - It provides clear separation and a unique layer between construction and representation of a specified object created by class.
  - It provides better control over construction process of the pattern created.
  - It gives the perfect scenario to change the internal representation of objects.

Prototype Design Pattern



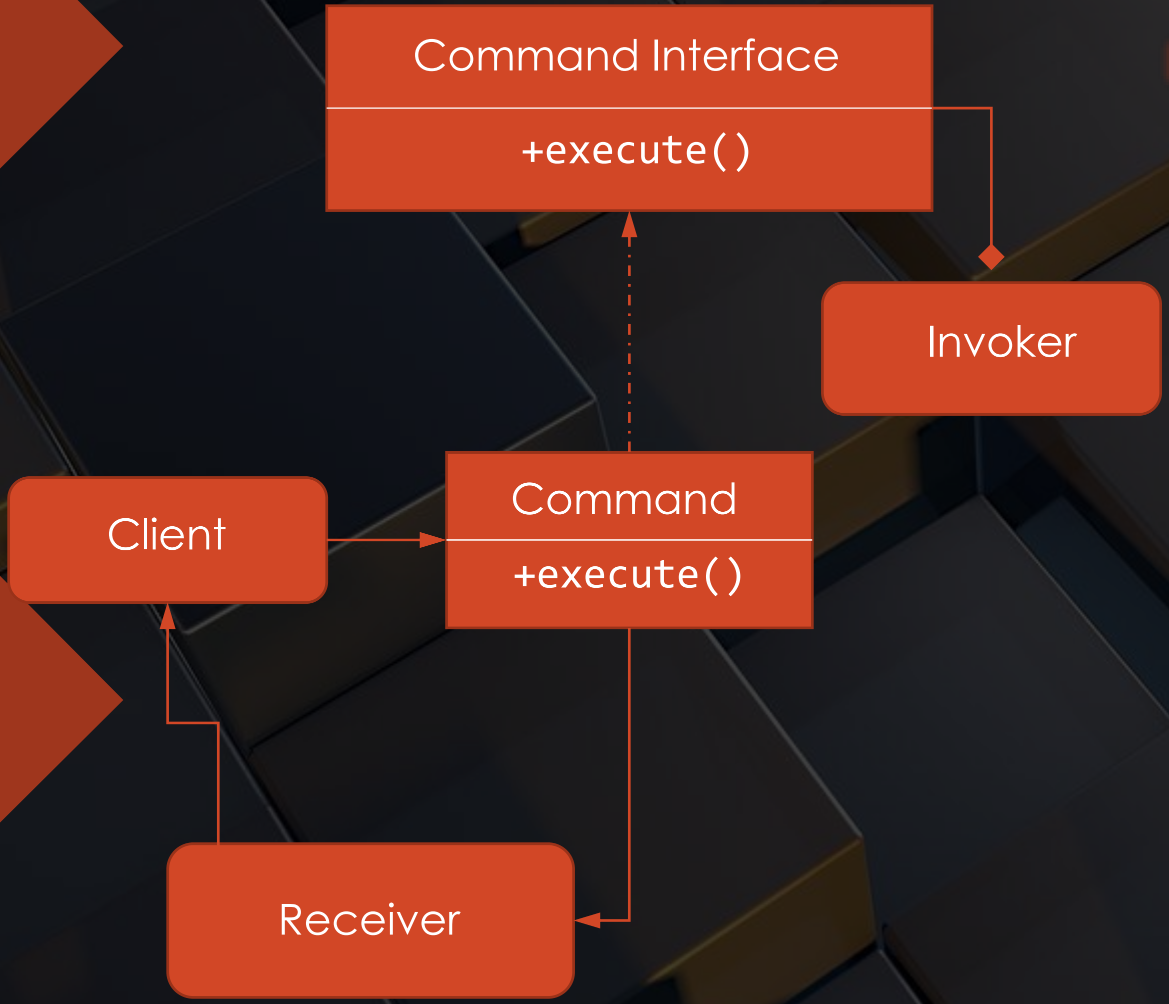
- Prototype design pattern helps to hide the complexity of the instances created by the class.
- The concept of the existing object will differ with that of the new object, which is created from scratch.
- The newly copied object may have some changes in the properties if required.
- This approach saves time and resources that go in for the development of a product.

Facade Design Pattern



- Facade design pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that any subsystem can use.
- A facade class knows which subsystem is responsible for a request.

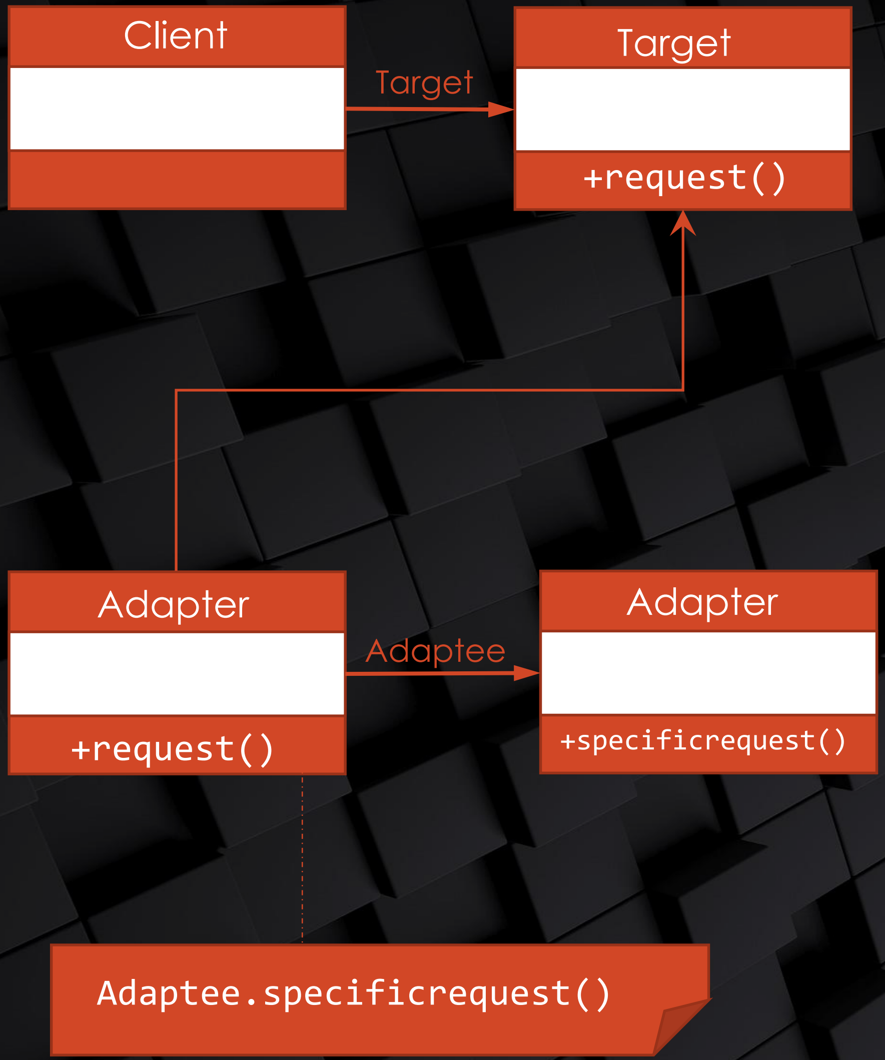
Command Design Pattern



- Command Pattern adds a level of abstraction between actions and includes an object, which invokes these actions.
- In this design pattern, client creates a command object that includes a list of commands to be executed.
- The command object created implements a specific interface.

Adapter Design Pattern

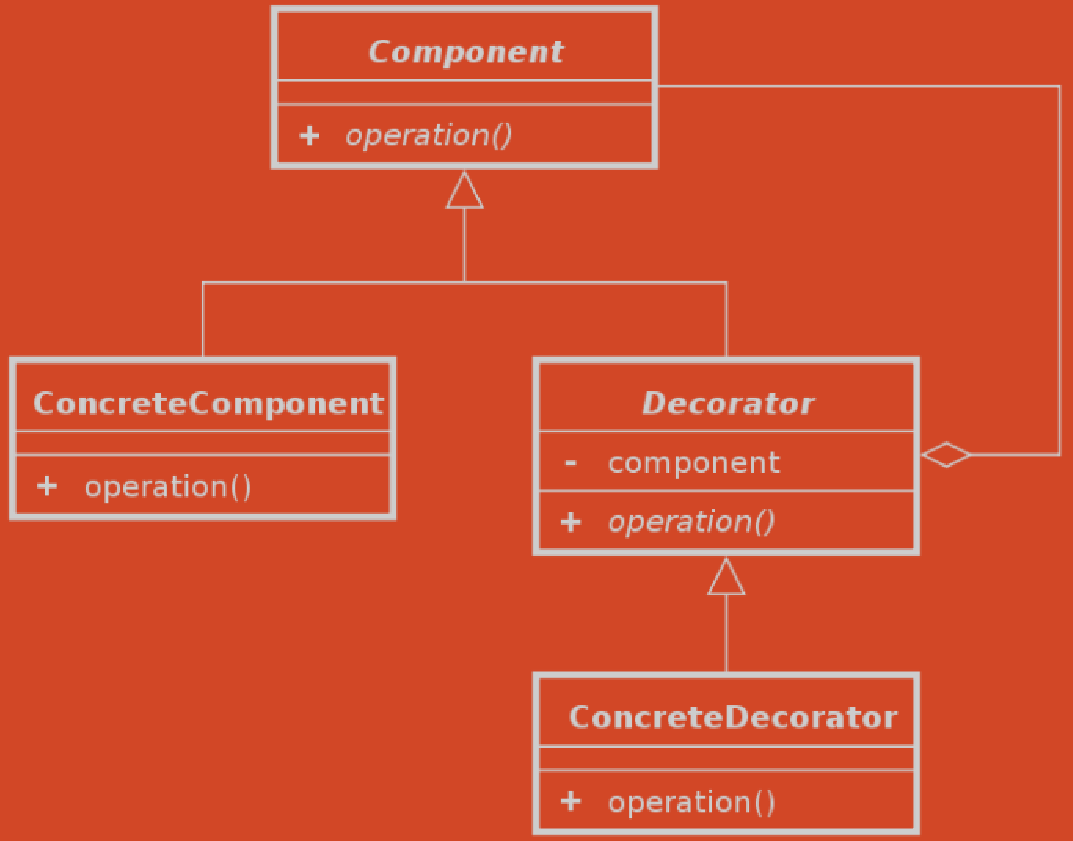
- Adapter pattern works as a bridge between two incompatible interfaces.
- This pattern involves a single class, which is responsible to join functionalities of independent or incompatible interfaces.
- A real life example could be the case of a card reader, which acts as an adapter between memory card and a laptop.
- You plug in the memory card into the card reader and the card reader into the laptop so that memory card can be read via the laptop.



Adapter Design Pattern

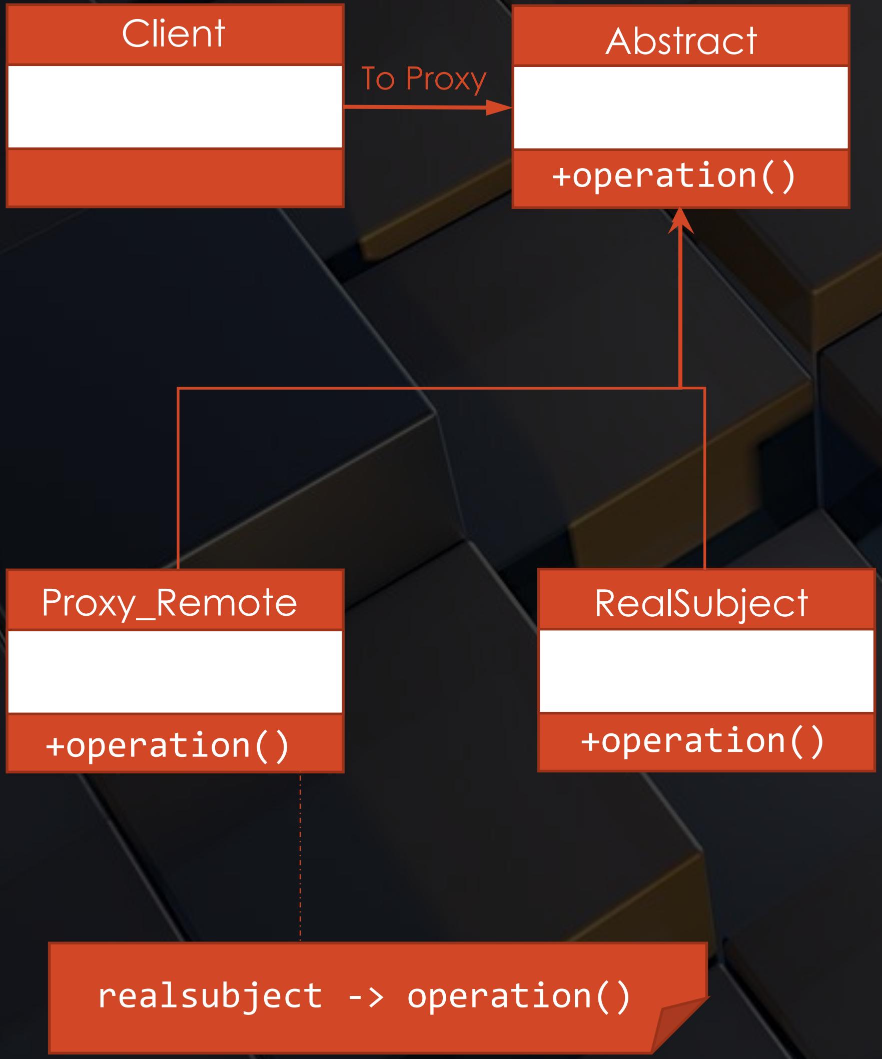
- The adapter design pattern helps to work classes together.
- It converts the interface of a class into another interface based on requirement.
- The pattern includes a speciation a polymorphism which names one name and multiple forms. Say for a shape class which can use as per the requirements gathered.
- There are two types of adapter patterns –
  - Object Adapter Pattern:
    - This design pattern relies on object implementation.
  - Class Adapter Pattern
    - This is an alternative way to implement the adapter design pattern.
    - The pattern can be implemented using multiple inheritances.

Decorator Design Pattern



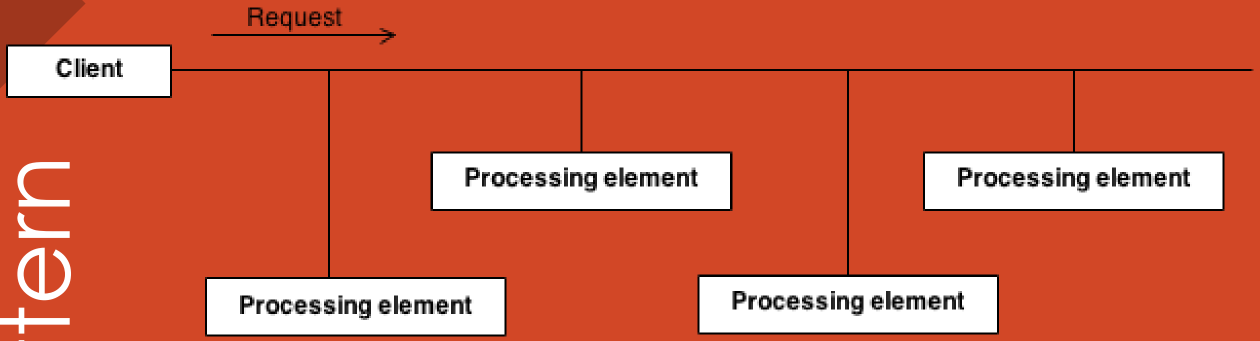
- Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This pattern creates a decorator class, which wraps the original class and provides additional functionality keeping the class methods signature intact.
- The motive of a decorator pattern is to attach additional responsibilities of an object dynamically.

Proxy Design Pattern



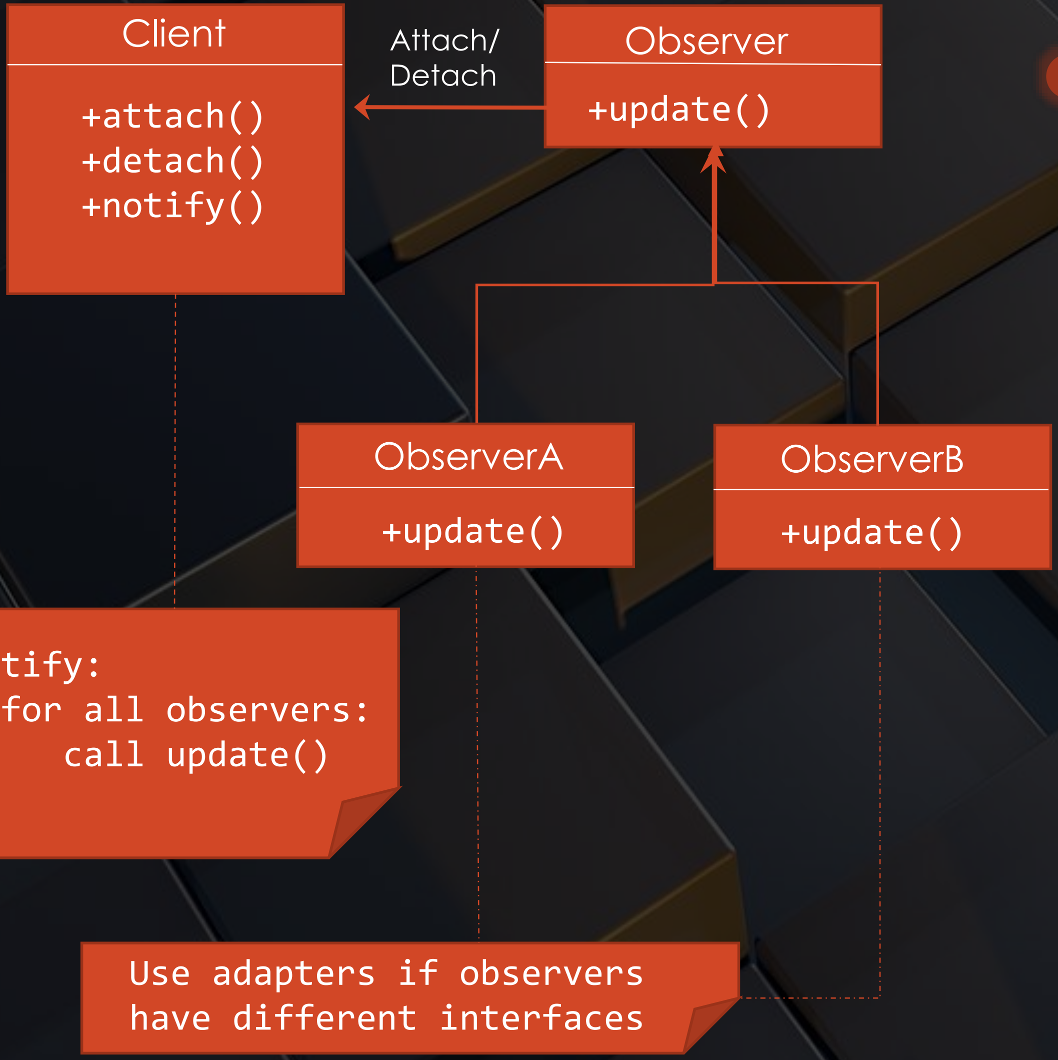
- The proxy design pattern includes a new object, which is called “Proxy” in place of an existing object which is called the “Real Subject”.
- The proxy object created of the real subject must be on the same interface in such a way that the client should have no idea that proxy is used in place of the real object.
- Requests generated by the client to the proxy are passed through the real subject.

Chain of Responsibility Pattern



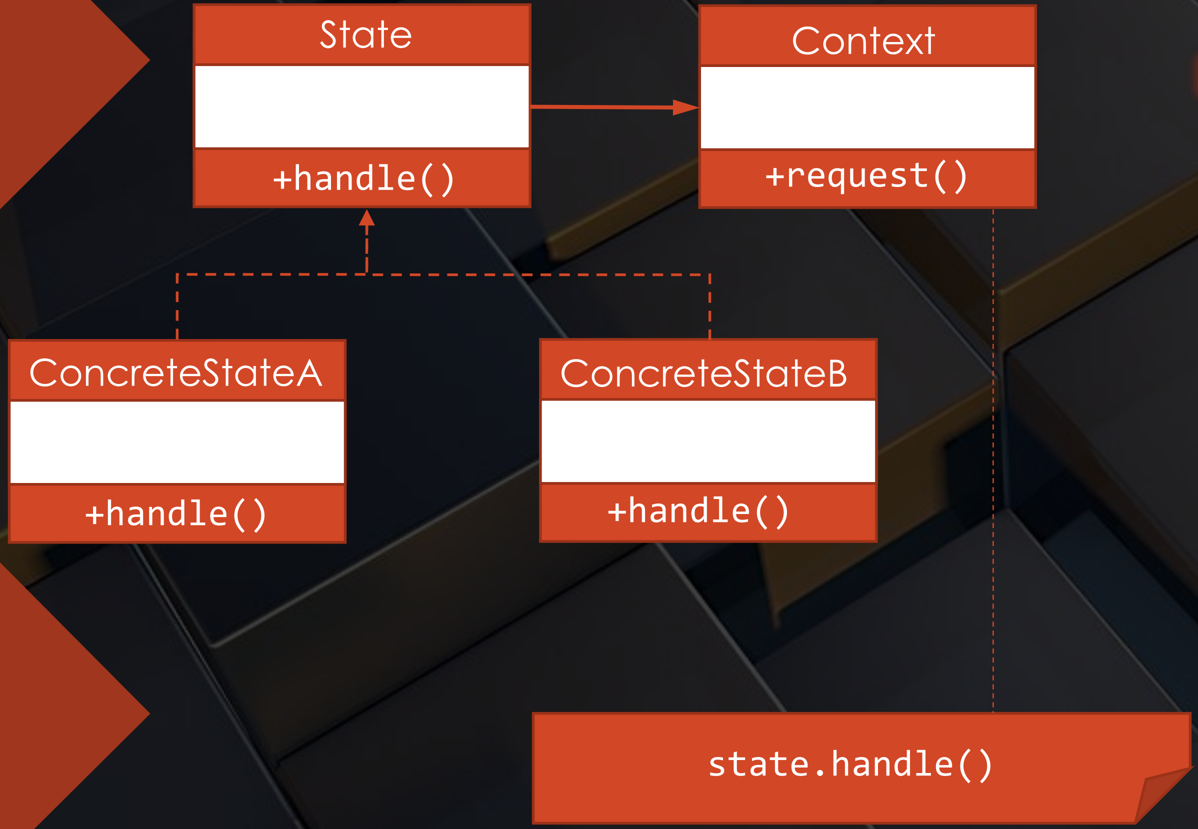
- The chain of responsibility pattern is used to achieve loose coupling in software where a specified request from the client is passed through a chain of objects included in it.
- It helps in building a chain of objects. The request enters from one end and moves from one object to another.
- This pattern allows an object to send a command without knowing which object will handle the request.

Observer Design Pattern



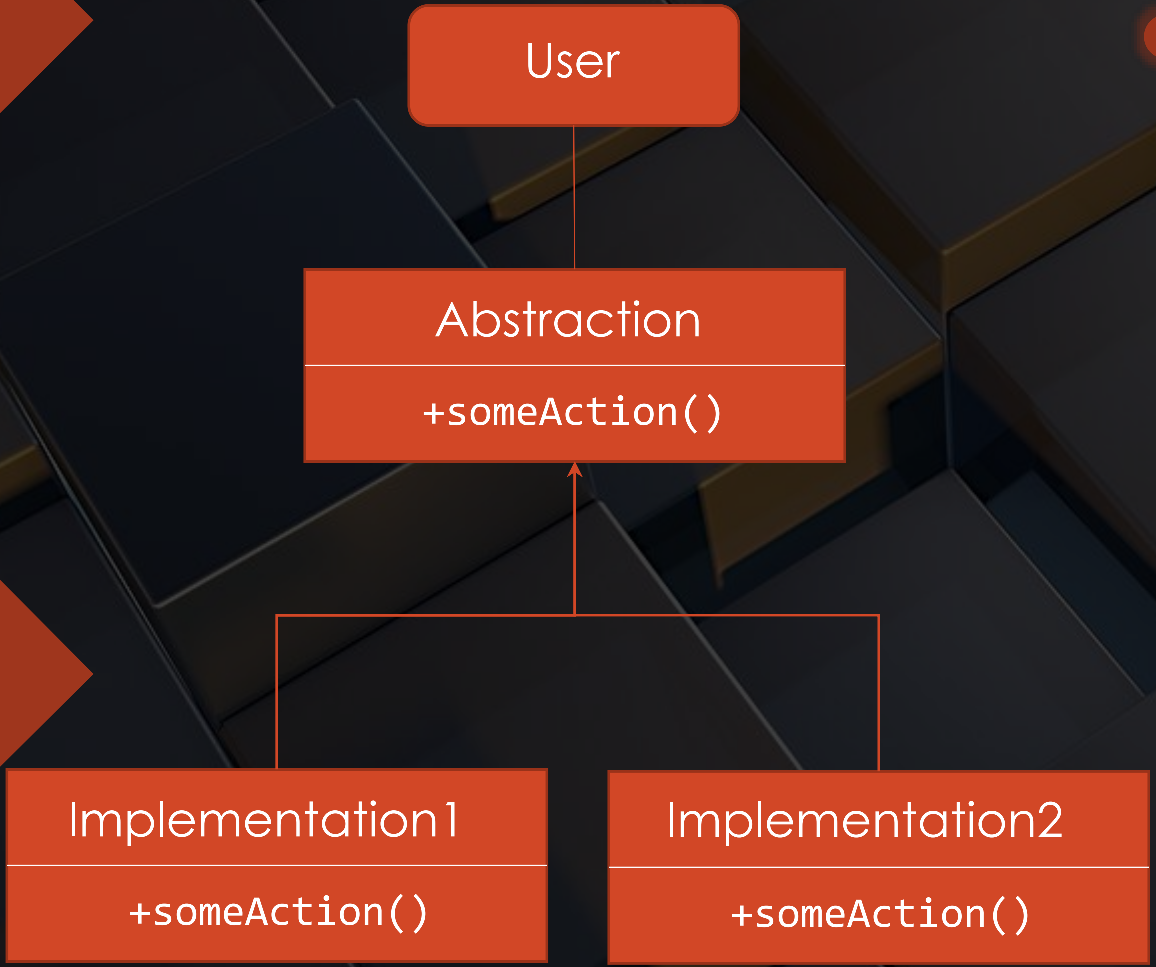
- In this pattern, objects are represented as observers that wait for an event to trigger.
- An observer attaches to the subject once the specified event occurs. As the event occurs, the subject tells the observers that it has occurred.

State Design Pattern



- It provides a module for state machines, which are implemented using subclasses, derived from a specified state machine class.
- The methods are state independent and cause transitions declared using decorators.

Strategy Design Pattern

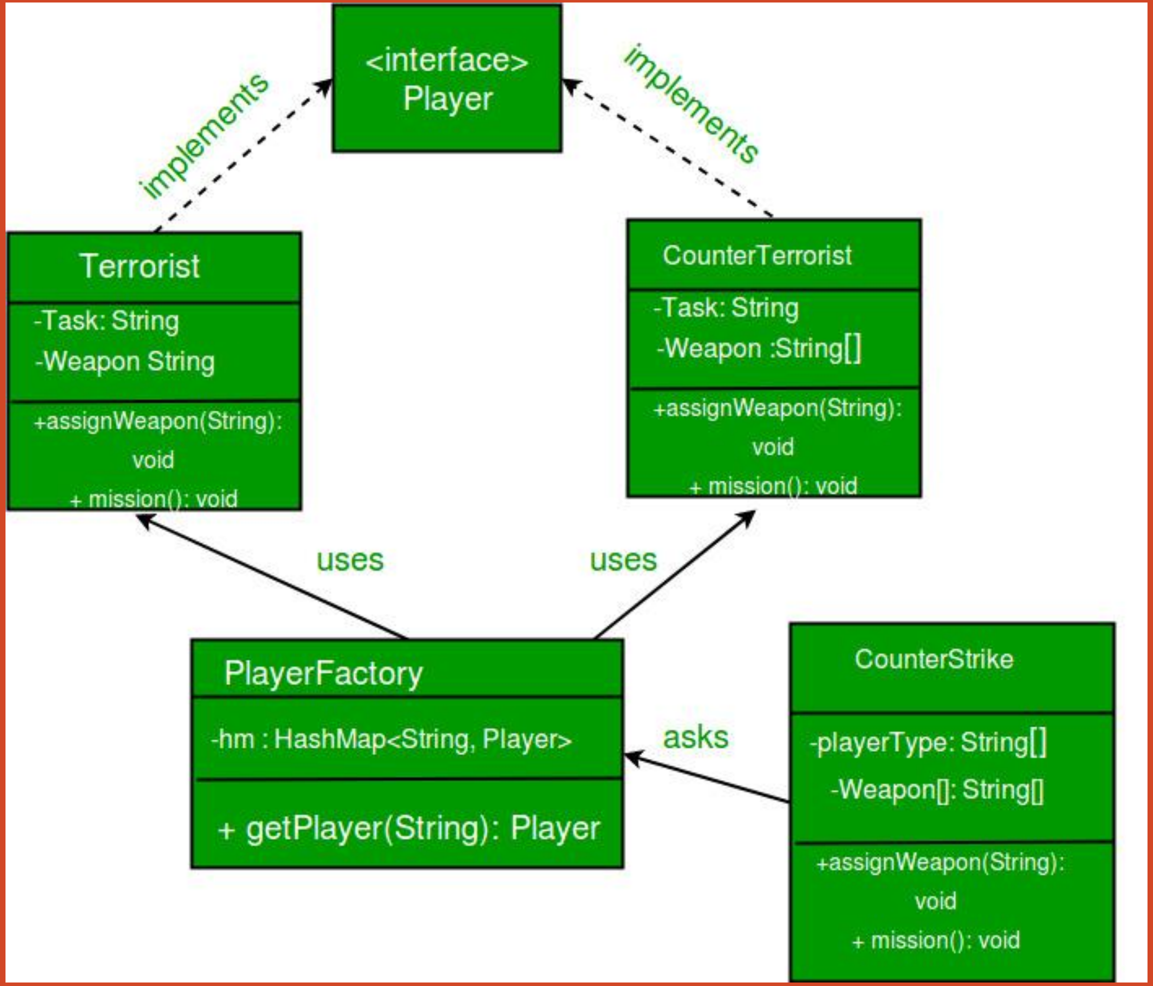


- The main goal of strategy pattern is to enable client to choose from different algorithms or procedures to complete the specified task.
- Different algorithms can be swapped in and out without any complications for the mentioned task.
- This pattern can be used to improve flexibility when external resources are accessed.

Template Design Pattern

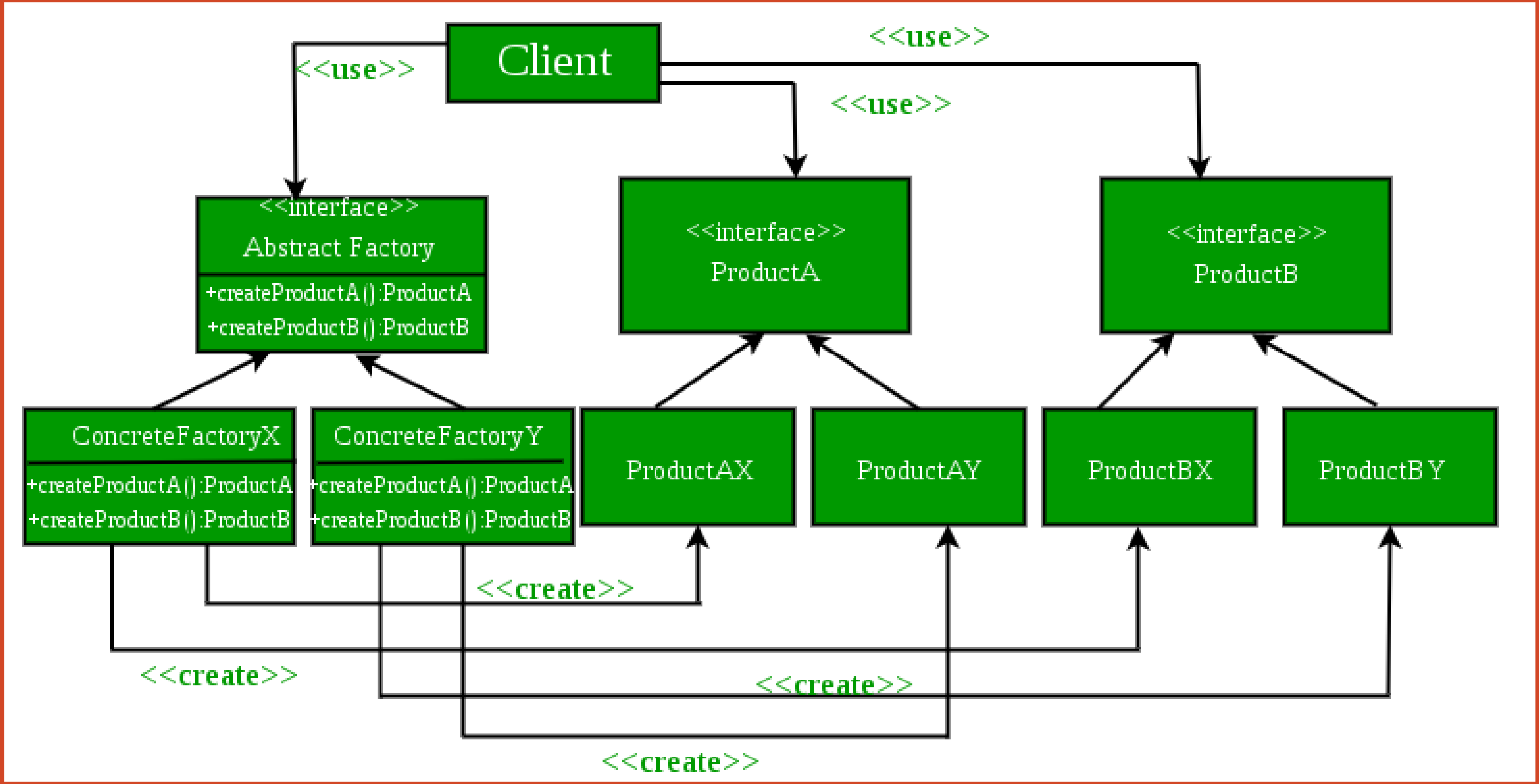
- A template pattern defines a basic algorithm in a base class using abstract operation where subclasses override the concrete behavior.
- The template pattern keeps the outline of algorithm in a separate method.
- This method is referred as the template method.
- The different features of the template pattern
  - It defines the skeleton of algorithm in an operation
  - It includes subclasses, which redefine certain steps of an algorithm.

Flyweight Design Pattern



- It provides a way to decrease object count.
- It includes various features that help in improving application structure.
- The most important feature of the flyweight objects is immutable.
- This means that they cannot be modified once constructed.
- The pattern uses a HashMap to store reference objects.

Abstract Factory Pattern



Abstract Factory Pattern

- The abstract factory pattern is also called factory of factories.
- This design pattern comes under the creational design pattern category.
- It provides one of the best ways to create an object.
- It includes an interface, which is responsible for creating objects related to Factory.

Iterator Design Pattern

- The iterator design pattern falls under the behavioral design patterns category.
- Developers come across the iterator pattern in almost every programming language.
- This pattern is used in such a way that it helps to access the elements of a collection (class) in sequential manner without understanding the underlying layer design.

Strings and Serialization

- String serialization is the process of writing a state of object into a byte stream.
- In python, the “pickle” library is used for enabling serialization.
- This module includes a powerful algorithm for serializing and de-serializing a Python object structure.
- “Pickling” is the process of converting Python object hierarchy into byte stream and “unpickling” is the reverse procedure.

Anti-Patterns

Features of Anti-Patterns:

1 Correctness:

These patterns literally break your code and make you do wrong things.

2 Maintainability

A program is said to be maintainable if it is easy to understand and modify as per the requirement. Importing module can be considered as an example of maintainability.

- Anti-patterns follow a strategy in opposition to predefined design patterns.
- The strategy includes common approaches to common problems, which can be formalized and can be generally considered as a good development practice.
- Usually, anti-patterns are opposite and undesirable.
- Anti- patterns are certain patterns used in software development, which are considered as bad programming practices.

Ramesh Sannareddy

Reach me

Linkedin

rsannareddy@gmail.com