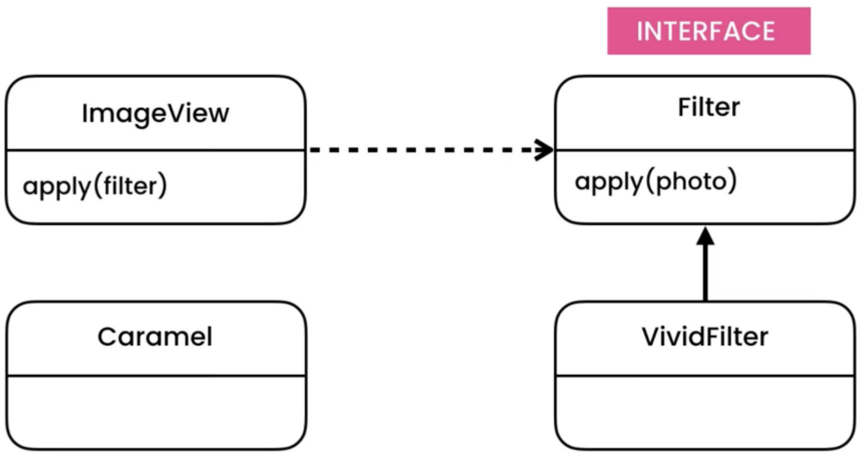
Structural Design Patterns

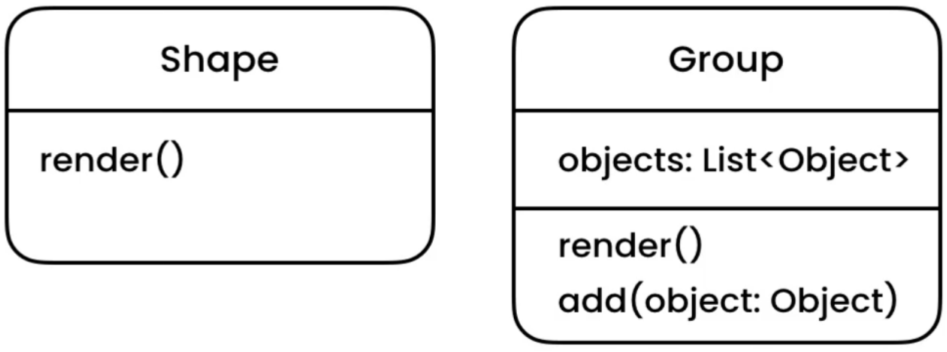
Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

When to use

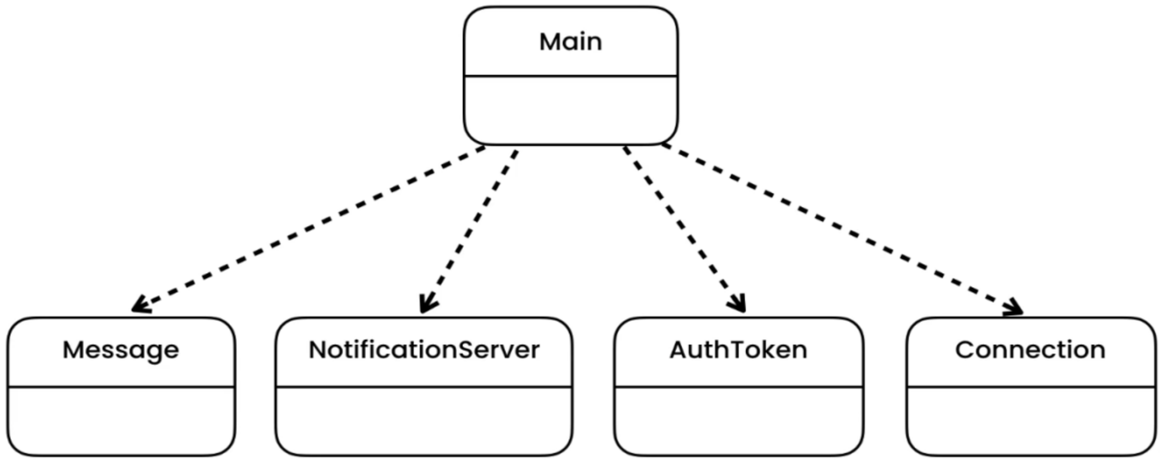
- **Adapter Pattern:** We have an existing class, we want to use it in our solution but the interface of that class does not have the form that we expect.
  For example, we want to use existing Caramel class but it does not implement our interface.



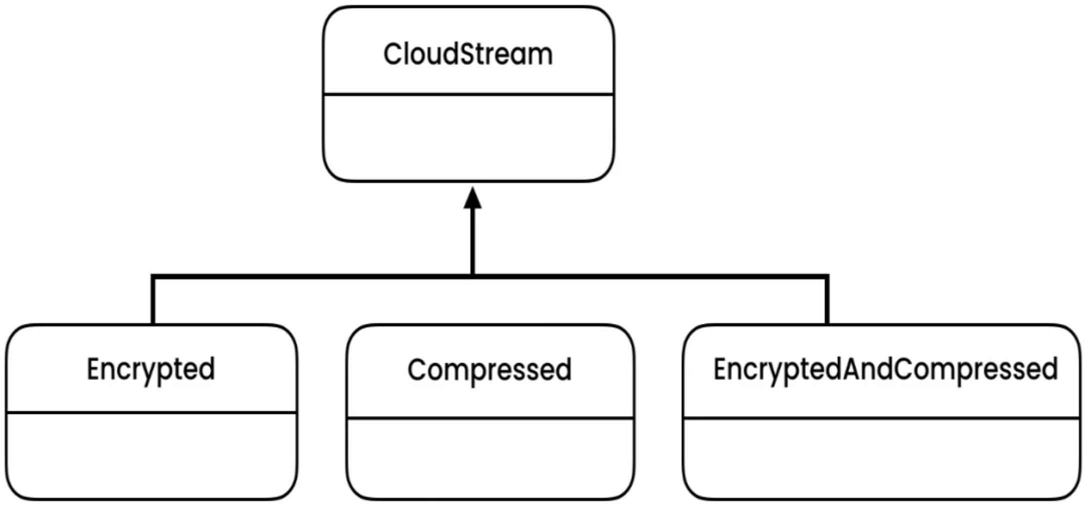
- **Composite Pattern:** When we want to treat the hierarchies in the same way whether they are container or parts but we always have to specify/cast their type to render the desired functionality because we are dealing with objects.
  For example, when we move our folder, we want all files inside to be moved too. Another scenario is, in some photoshop, when we select multiple shapes and want to move them, we want to treat the shapes and group of shapes in the same way but in the code, we have to explicitly cast to the shape or group class while calling render().



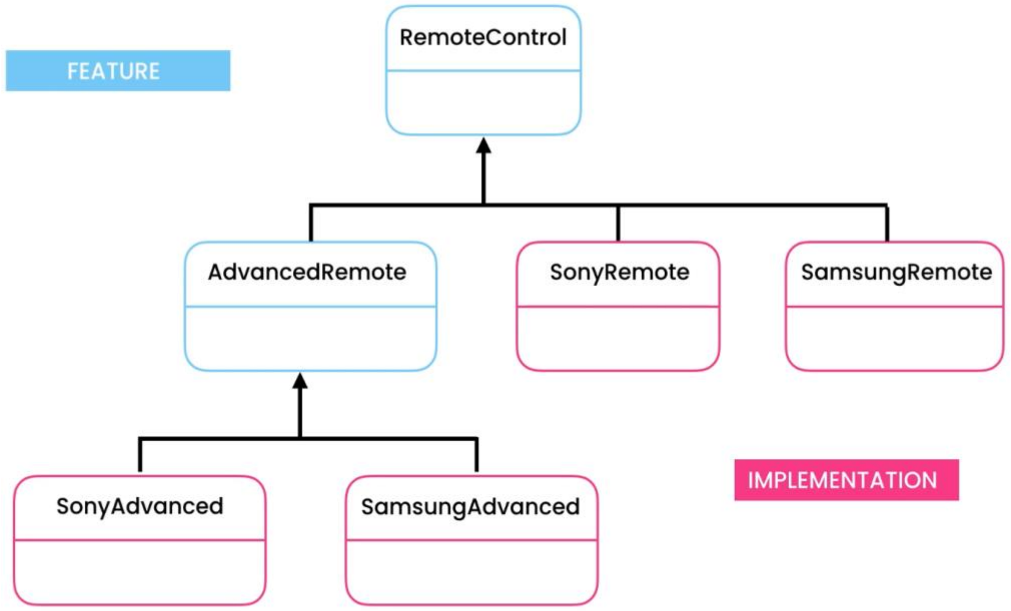
- **Facade Pattern:** When there is too much coupling of classes with client and adding/modifying a class cause breaking changes in the client.
  For example, we want to implement a notification server, which performs connection, authentication, sending message and disconnection but they are called by client and modification of these classes result in breaking changes in the client.



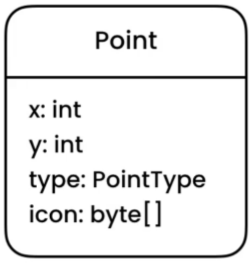
- **Decorator Pattern:** When we need to add additional behavior to an object.
  For example, we want to implement cloud stream, but later we want to do some manipulation on data (encryption) before writing it on the stream. Later, we decide to add additional behavior e.g. compression before encryption. But sometimes we want to make it optional and want to give the control to the client whether he wants encryption, compression or both in any order.



- **Bridge Pattern:** When multiple hierarchies grow in different directions but we end up adding many classes to implement these hierarchies.
  For example, we have two types of remote controls i.e. basic and advanced. Then we have multiple implementations of remote controls i.e. Sony Samsung, LG which can be basic or advanced. And we have to add SonyBasic, SonyAdvanced, SamsungBasic, SamsungAdvanced, LGBasic and LGAdvanced classes which is not modular.



- **Flyweight Pattern:** It is used when we need to create a large number of similar objects and to minimize memory usage by sharing as much data as possible.
  For example, we need to implement a point object which has x,y coordinates, type and icon. Suppose the Point.PointType = café and it has 100 objects, so we will save 100 instaces of same café icon which will consume a lot of memory.



- **Proxy Pattern:** Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

  For example, we have a massive object that consumes a vast amount of system resources. We need it from time to time, but not always. We load all the resources first to access only one object.