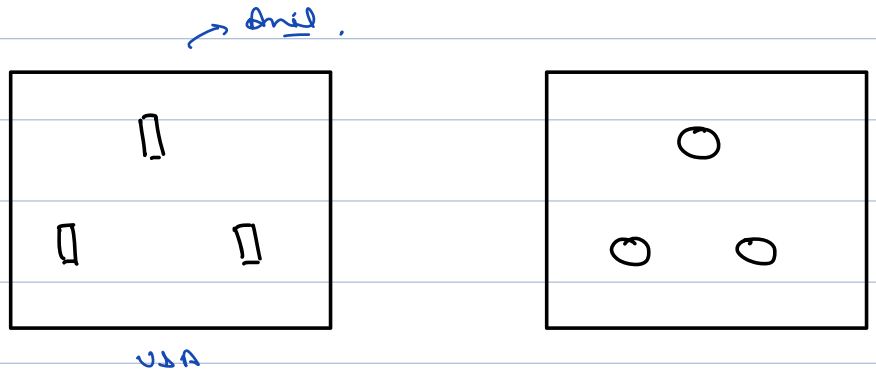
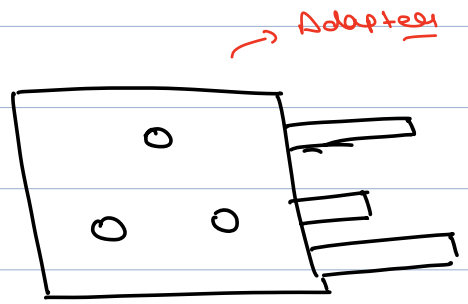
Structural Patterns

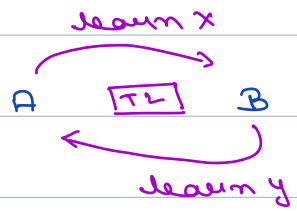
↳ Organising & structuring classes and objects to create more flexible efficient & maintenable code.

- Adapter
- facade
- Decorator
- flyweight

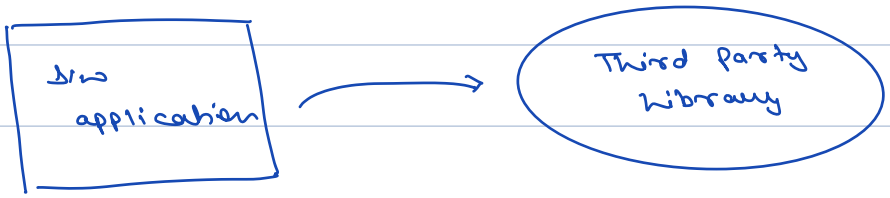
Adapter

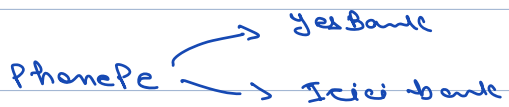


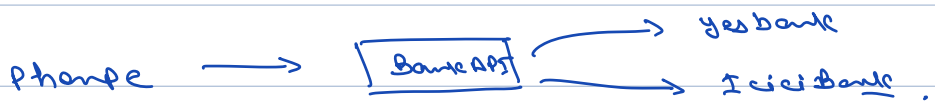




Adapter :- Intermediate layer which converts one form to another.







PhonePe

```
YesBankAPI yb = new YesBankAPI();

yb.getBalance();

yb.transferMoney();
```

ICICI Bank API
↓
checkBalance()
moneyTransfer()

Codebase is directly dependent on 3rd party class.

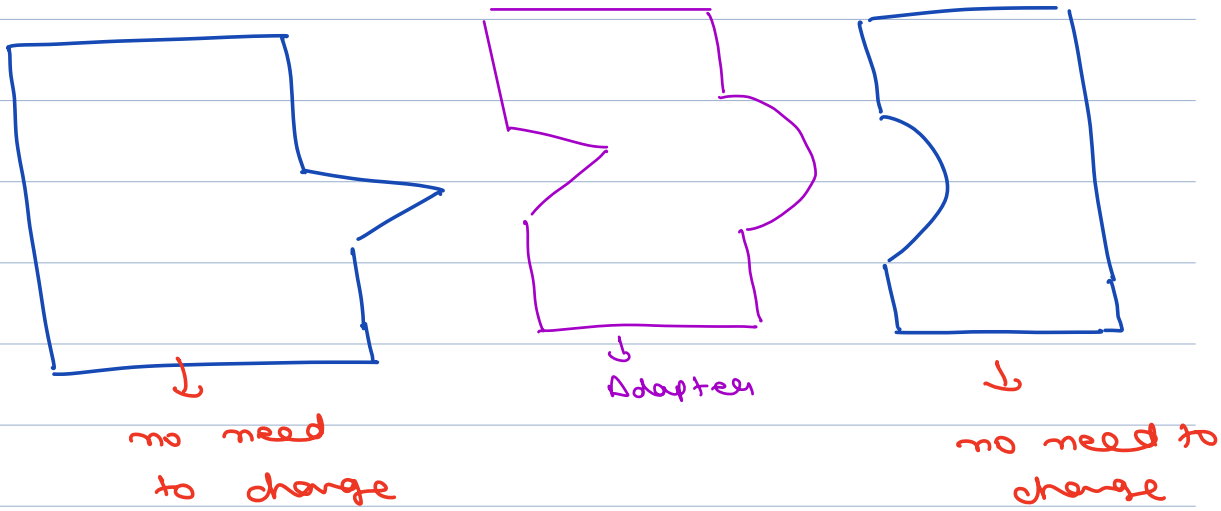
Dependency Inversion

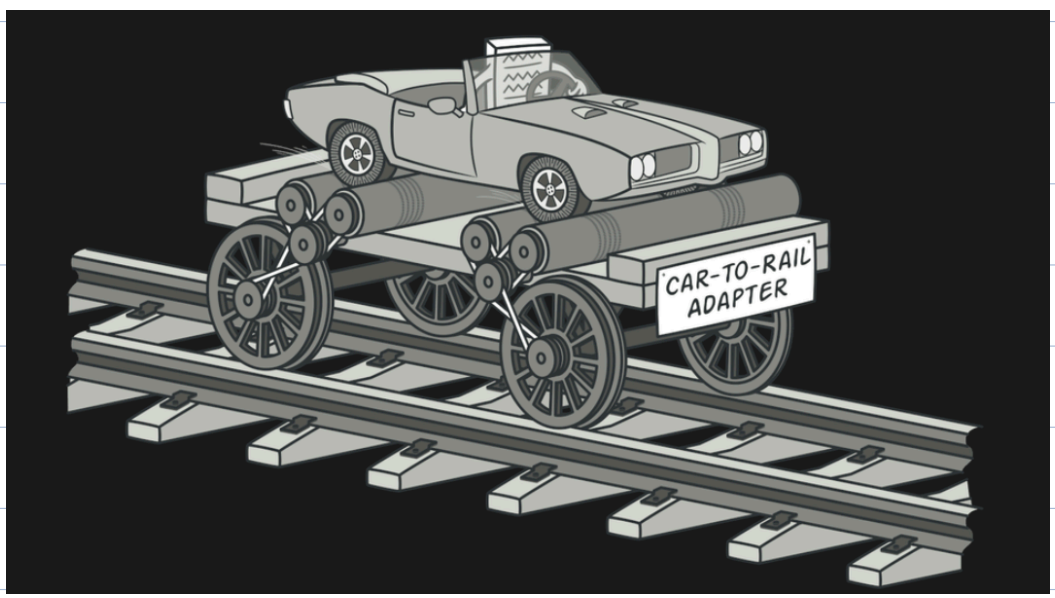
client ⟶ 3rd Party API

client ⟶ Interface

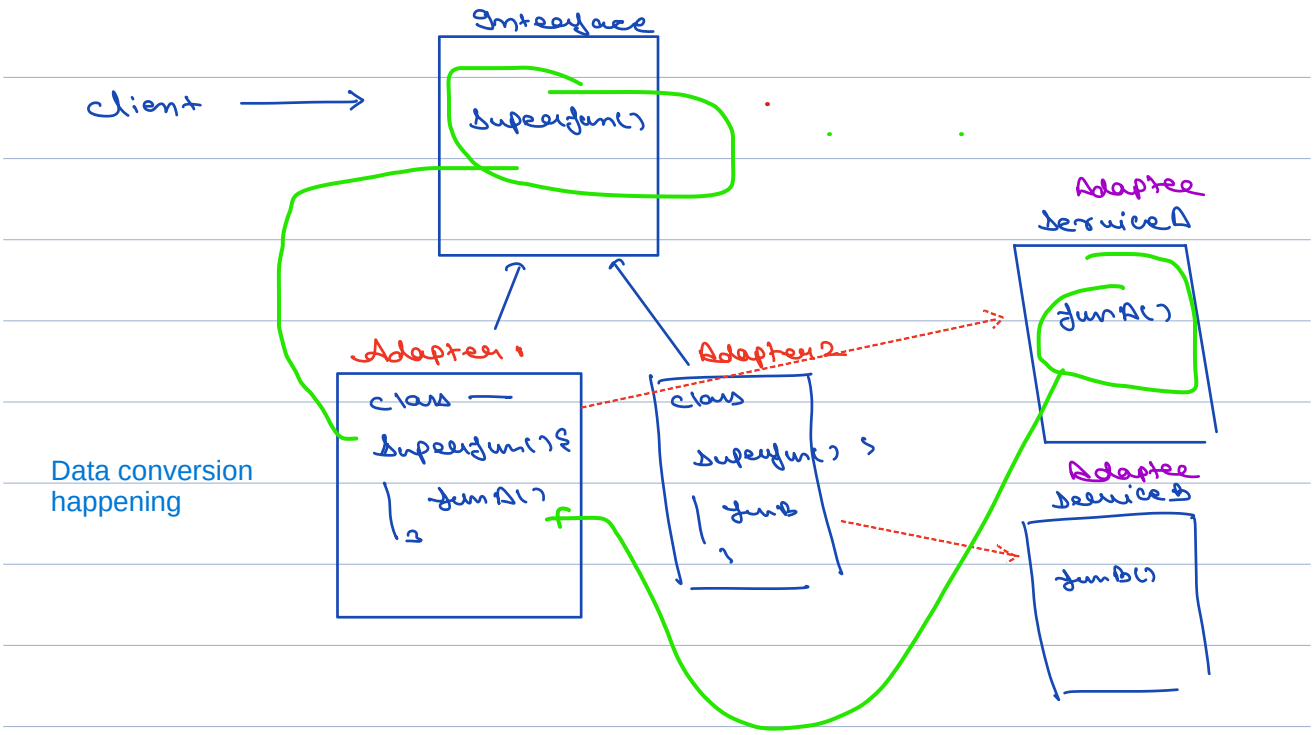
↑

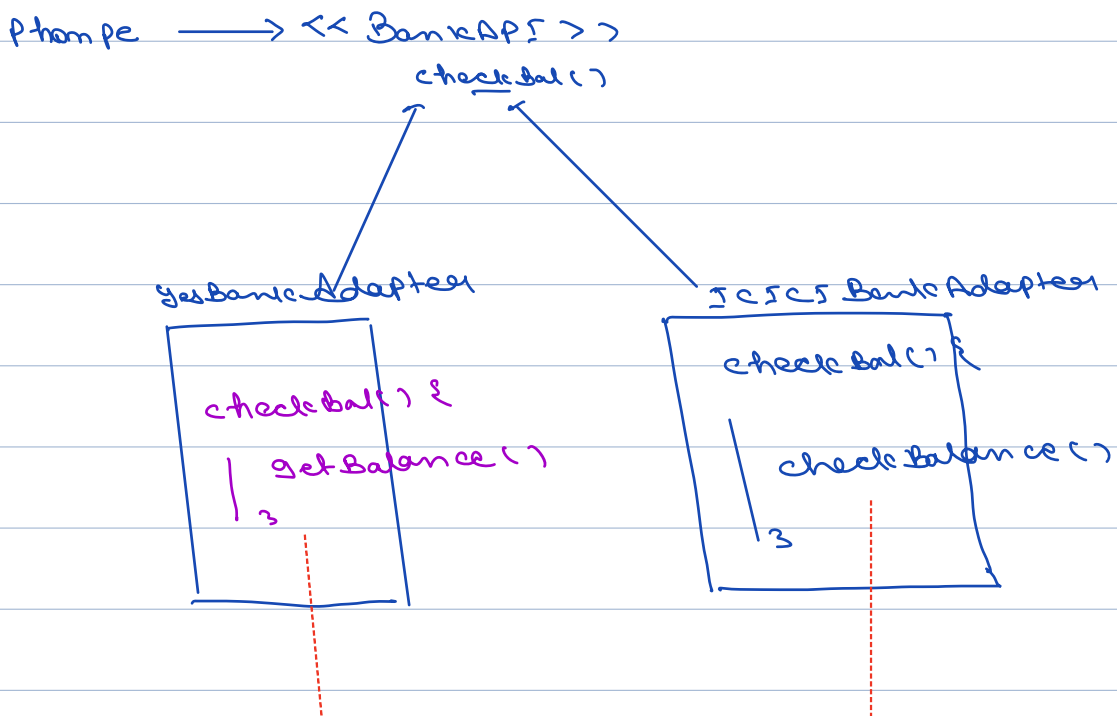
3rd Party API

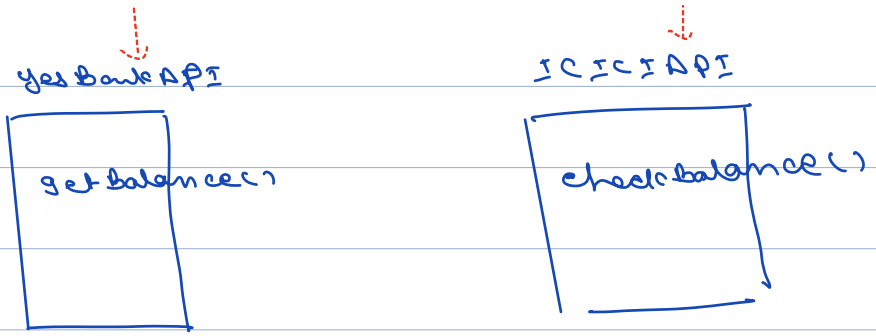

Adapter
no need to change
no need to change


CAR-TO-RAIL ADAPTER

client ko implementation interface k methods m chahiye but koi bhi service(phonePe) apne khud k mtds rkhegi... to koi bhi bank ka interface y force kaise kr skta h.. ki phonePe apna poora codebase us particular bank k according kre.. kl se doosra bank bhi same bola to.. so here comes Adapter






yesBankAPI
getBalance()
ICICIAPI
checkBalance()

Where this Adapter can be used

→ Interface compatibility

→ Legacy code.

→ Third party integration.

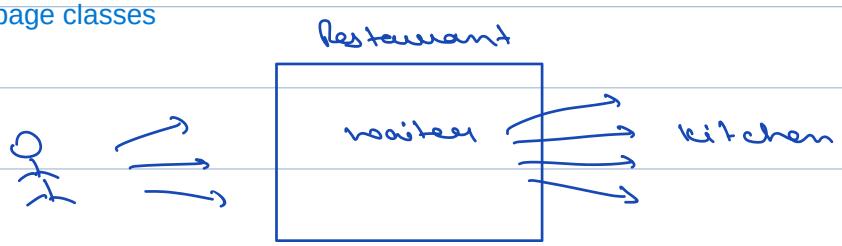
→ Cross platform compatibility.

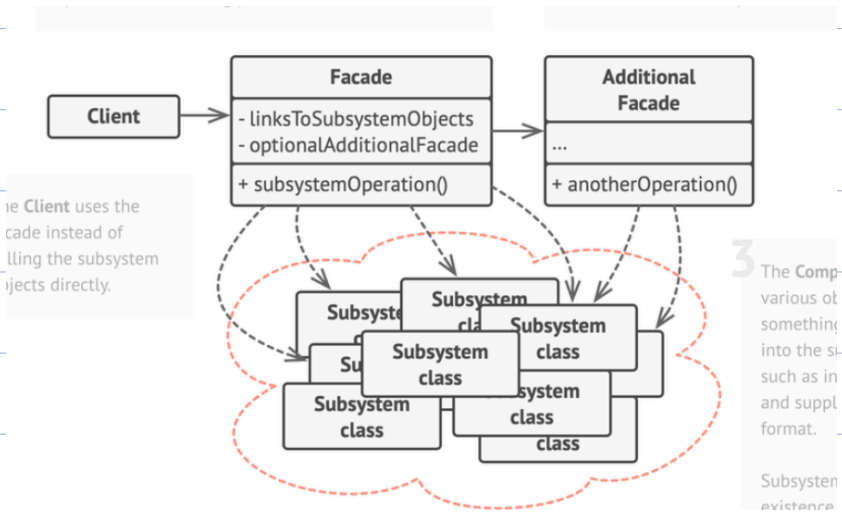
facade Design Pattern

↳ Deceptive outward appearance

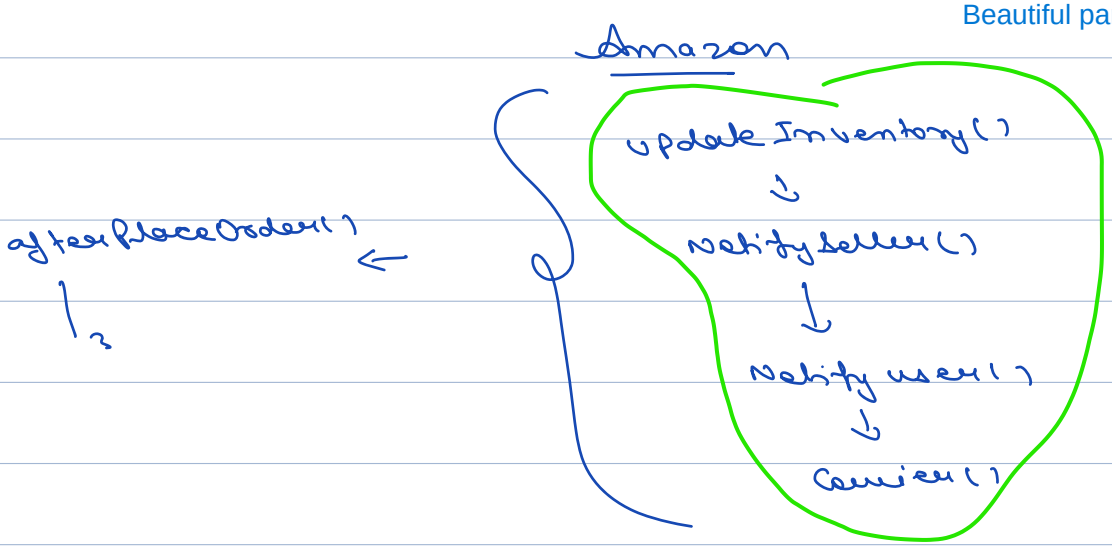
just google for meaning

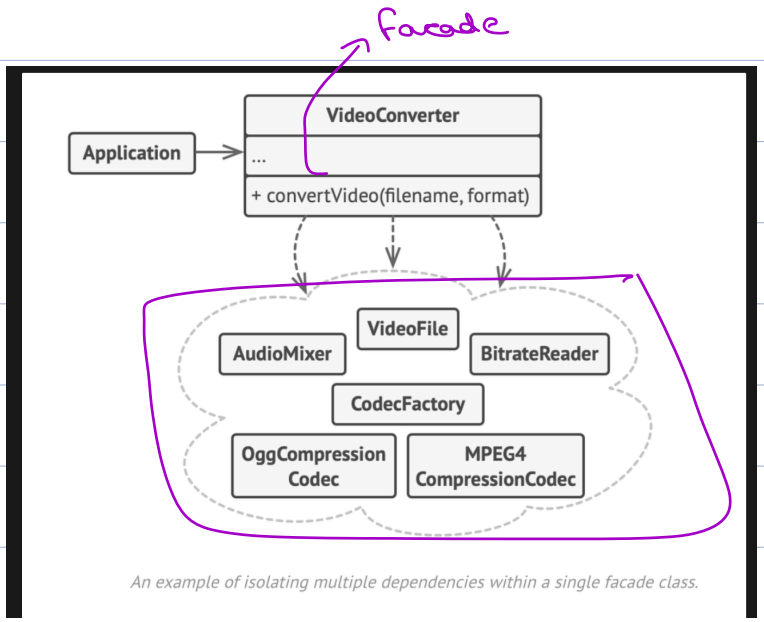
background ka code expose nhi krna.. bs front dikhana h.. jo beautiful ho.. e.g. apni test classes.. instead of page classes





Beautiful part





An example of isolating multiple dependencies within a single facade class.

```
// These are some of the classes of a complex 3rd-party video
// conversion framework. We don't control that code, therefore
// can't simplify it.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...
```

```
// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

Just this video converter is facade

Often, subsystems get more complex over time. Even applying design patterns typically leads to creating more classes. A subsystem may become more flexible and easier to reuse in various contexts, but the amount of configuration and boilerplate code it demands from a client grows ever larger. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.

facade kuch jyada hi simple ke dega .. ab user na to video format na hi bitrate.. or other things change kr skta h..

There is a tradeoff between simplification and restriction. Over-simplifying a system means that the developer is over-restricted, therefore less freedom than necessary which not always a good thing. Under-simplifying Facade pattern means that there is too much freedom which makes the Facade pattern irrelevant. Finding the fine balance is what makes a good, useful and effective Facade pattern.

```
public class CarEngineFacade {
    private static int DEFAULT_COOLING_TEMP = 90;
    private static int MAX_ALLOWED_TEMP = 50;
    private FuelInjector fuelInjector = new FuelInjector();
    private AirFlowController airFlowController = new AirFlowController();
    private Starter starter = new Starter();
    private CoolingController coolingController = new CoolingController();
    private CatalyticConverter catalyticConverter = new CatalyticConverter();

    public void startEngine() {
        fuelInjector.on();
        airFlowController.takeAir();
        fuelInjector.on();
        fuelInjector.inject();
        starter.start();
        coolingController.setTemperatureUpperLimit(DEFAULT_COOLING_TEMP);
        coolingController.run();
        catalyticConverter.on();
    }

    public void stopEngine() {
        fuelInjector.off();
        catalyticConverter.off();
        coolingController.cool(MAX_ALLOWED_TEMP);
        coolingController.stop();
        airFlowController.off();
    }
```

Adapter :- It is about making two interface which are not compatible. Compatible with each other. makes Objects Compatible

Facade :- It's about taking a bunch of complex objects and interactions and creating a facade in front of it, instead of dealing with

that complexity.

Hides complex logics.

Hides complex logics