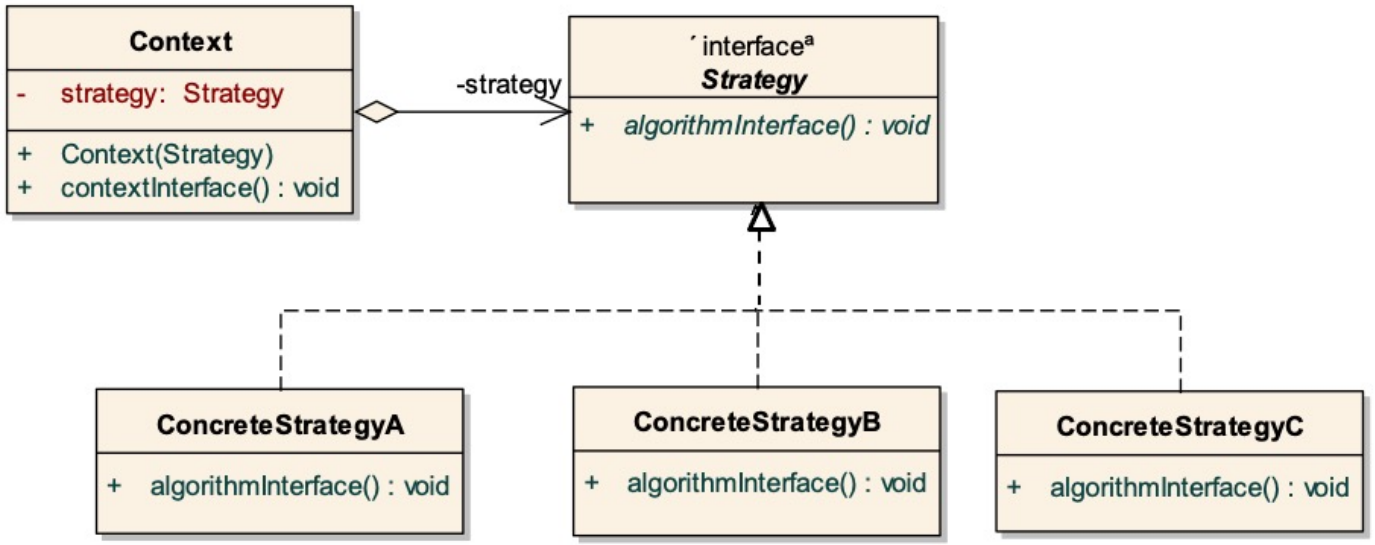
# Design patterns for web development

Software Engineering 2

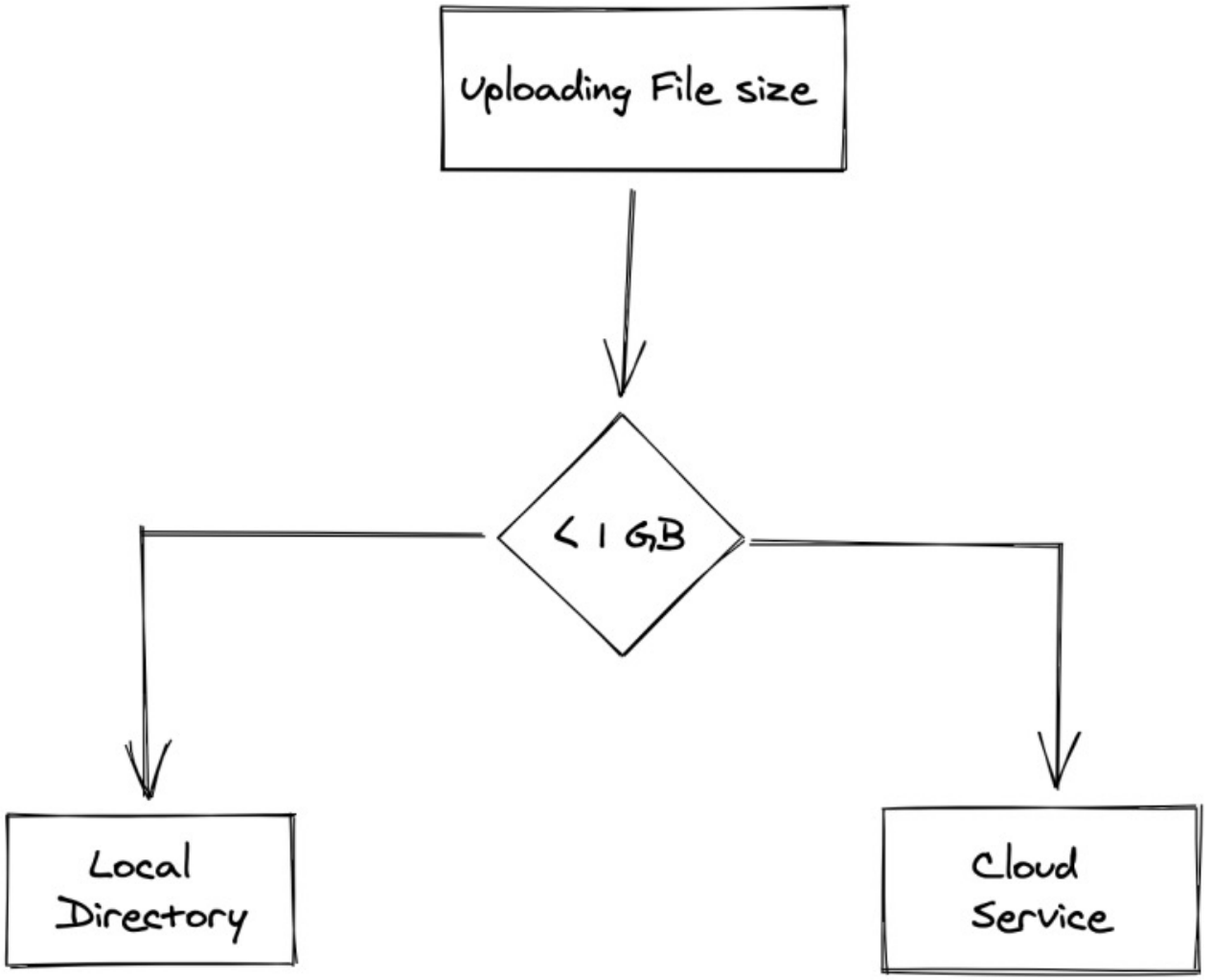
# 1. Strategy Pattern

- The strategy is pattern is like an advanced version of an if else statement.
- It's basically where you make an interface for a method you have in your base class.
- This interface is then used to find the right implementation of that method that should be used in a derived class (at run-time).

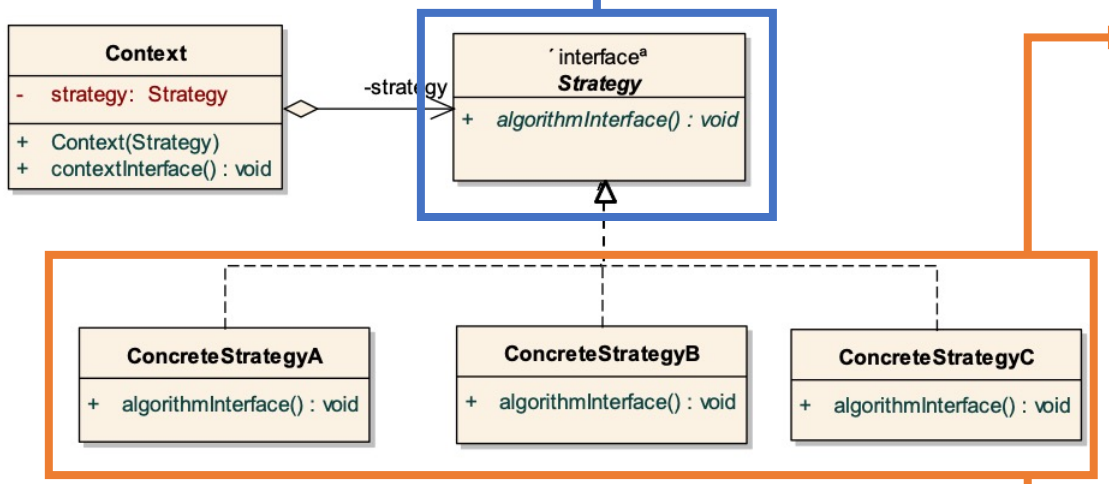


# Strategy Pattern – Example 1

- Consider that you want to handle file storage based on file size in your application:



- Map example codes to the Strategy Pattern template

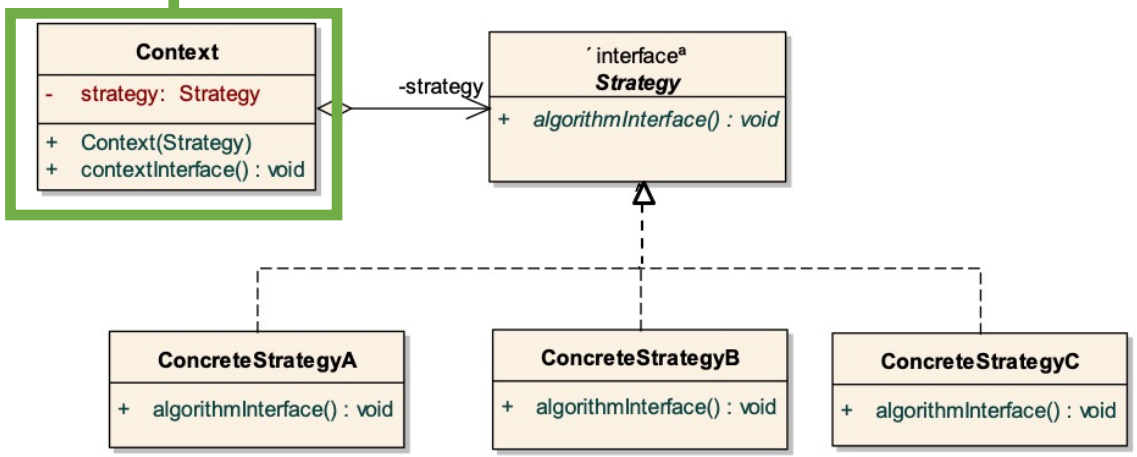



```typescript
export default interface IFileWriter {
    write(filepath: string | undefined) : boolean
}
```

```typescript
import IFileWriter from '../interface/IFileWriter'

export default class AWSWriterWrapper implements IFileWriter {

    write() {
        console.log("Writing File to AWS S3")
        return true
    }
}
```

```typescript
import IFileWriter from '../interface/IFileWriter'

export default class DiskWriter implements IFileWriter {

    write(filepath : string) {
        console.log("Writing File to Disk",filepath)
        return true
    }
}
```

- Map example codes to the Strategy Pattern template




```
import IFileWriter from '../interface/IFileWriter'

export default class Writer {

    protected writer
    constructor(writer: IFileWriter) {
        this.writer = writer
    }

    write(filepath : string) : boolean {
        return this.writer.write(filepath)
    }
}
```

```
let size = 1000

            if(size < 1000){
                const writer = new Writer(new DiskFileWriter())
                writer.write("file path comes here")
            }
            else{
                const writer = new Writer(new AWSFileWriter())
                writer.write("writing the file to the cloud")
            }
```

# Strategy Pattern - Example 2

- A Shopping cart that lets customers check out with various payment methods.
- The strategy design pattern lets us decouple the payment methods from the checkout process
- So, we can add or update strategies without changing any code in the shopping cart or checkout process.

# “PaymentMethodStrategy.ts”

- A single class with multiple static methods.
- Each method takes the same parameter “*customerInfo*” that parameter has a defined type of *customerInfoType*.
- Note that each method has its own implementation and uses different values from the *customerInfo*.
- With the strategy pattern, you can also dynamically change the strategy being used at run time.

```
class PaymentMethodStrategy {

  const customerInfoType = {
    country: string
    emailAddress: string
    name: string
    accountNumber?: number
    address?: string
    cardNumber?: number
    city?: string
    routingNumber?: number
    state?: string
  }

  static BankAccount(customerInfo: customerInfoType) {
    const { name, accountNumber, routingNumber } = customerInfo
    // do stuff to get payment
  }

  static BitCoin(customerInfo: customerInfoType) {
    const { emailAddress, accountNumber } = customerInfo
    // do stuff to get payment
  }

  static CreditCard(customerInfo: customerInfoType) {
    const { name, cardNumber, emailAddress } = customerInfo
    // do stuff to get payment
  }

  static MailIn(customerInfo: customerInfoType) {
    const { name, address, city, state, country } = customerInfo
    // do stuff to get payment
  }

  static PayPal(customerInfo: customerInfoType) {
    const { emailAddress } = customerInfo
    // do stuff to get payment
  }
}
```

# “config.json”

- You can also set a default implementation in a simple *config.json* file like this:

```
{
  "paymentMethod": {
    "strategy": "PayPal"
  }
}
```

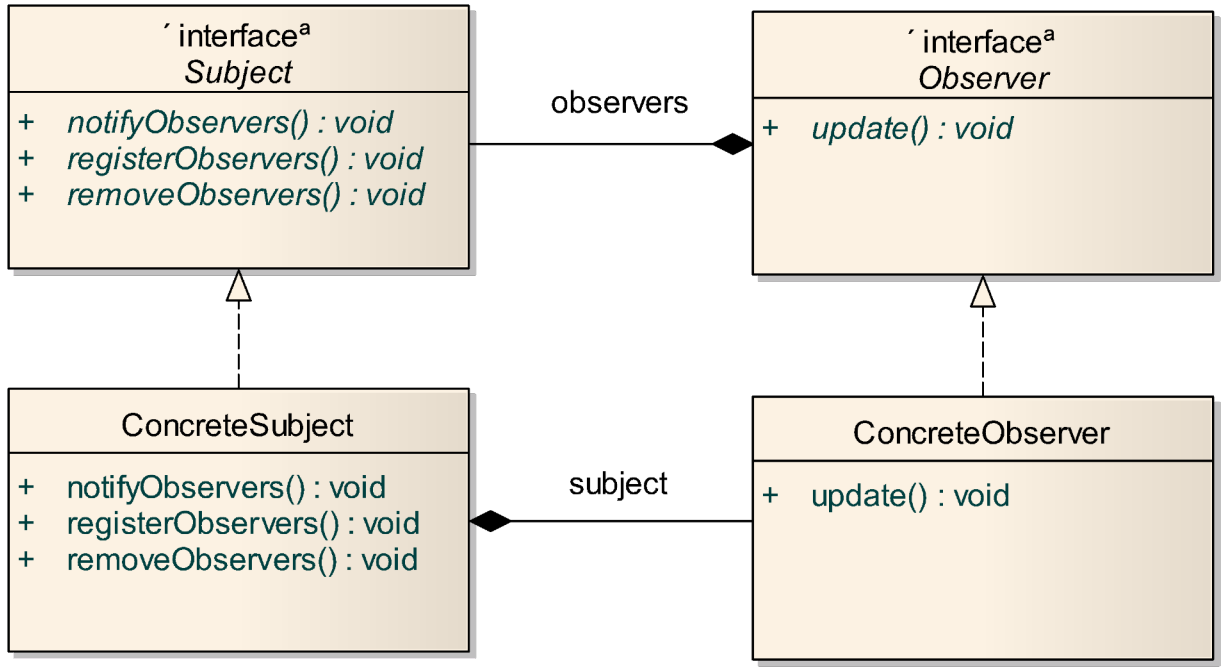
# “Checkout.ts”

- Import a couple of files so we have the payment method strategies available and the default strategy from the *config*.
- Create the class with the constructor and a fallback value.
- The *changeStrategy* method is for the ability to change the payment strategy
- The payment strategy immediately based on their inputs, and it dynamically sets the *strategy* before the payment is sent for processing.

```
const PaymentMethodStrategy = require('./PaymentMethodStrategy')
const config = require('./config')

class Checkout {
  constructor(strategy='CreditCard') {
    this.strategy = PaymentMethodStrategy[strategy]
  }

  // do some fancy code here and get user input and payment method

  changeStrategy(newStrategy) {
    this.strategy = PaymentMethodStrategy[newStrategy]
  }

  const userInput = {
    name: 'Malcolm',
    cardNumber: 3910000034581941,
    emailAddress: 'mac@gmailer.com',
    country: 'US'
  }

  const selectedStrategy = 'Bitcoin'

  changeStrategy(selectedStrategy)

  postPayment(userInput) {
    this.strategy(userInput)
  }
}

module.exports = new Checkout(config.paymentMethod.strategy)
```

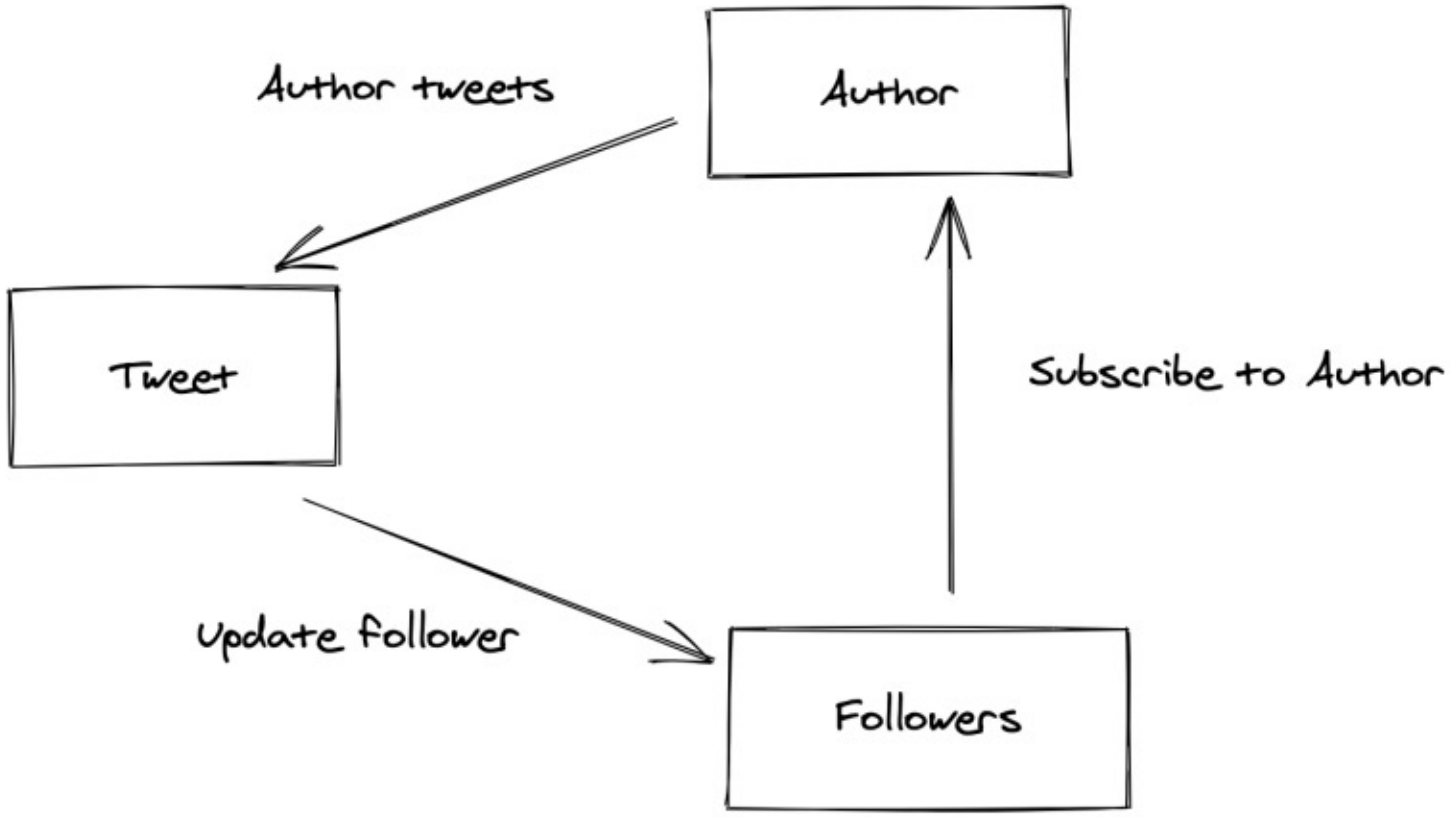
# 2. Observer Pattern

- If you've ever used the MVC pattern,
  - The Model part is like a subject
  - The View part is like an observer of that subject
- The goal is to create this one-to-many relationship between the subject and all of the observers waiting for data so they can be updated.



# Observer Pattern – Example

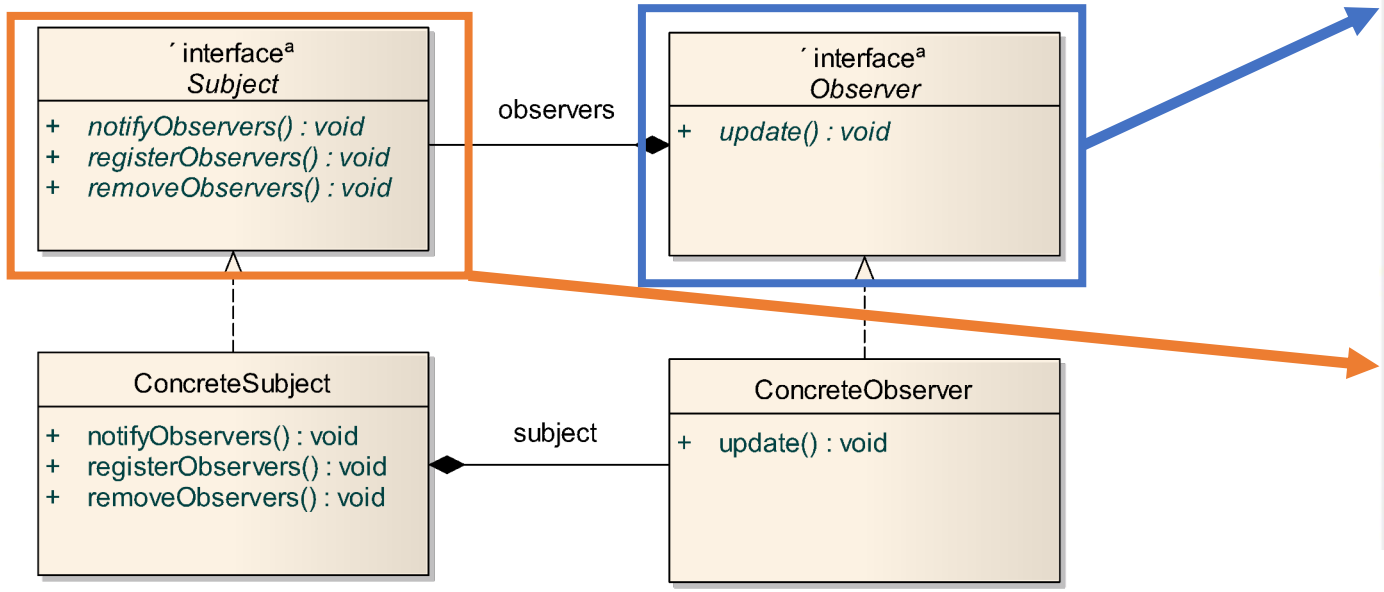
- Here, we have
  - Author
  - Tweet
  - Follower
- Followers can subscribe to the Author.
- Whenever there's a new Tweet, the follower is updated.



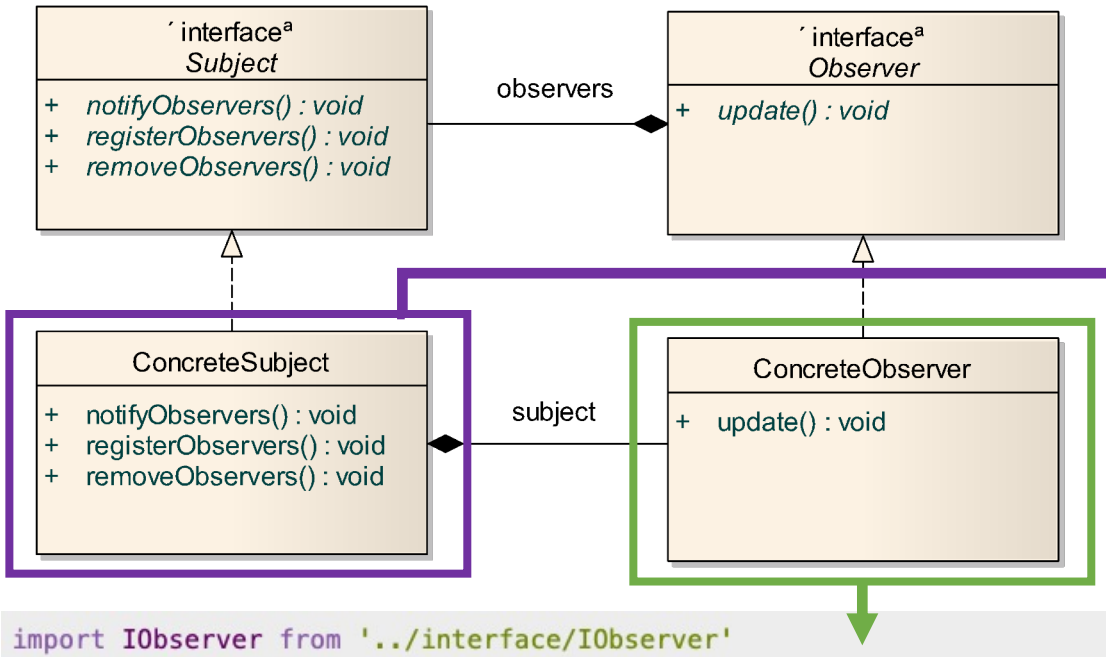
“Tweet.ts”

```
export default class Tweet {

    message : string
    author: string

    constructor(message : string,author: string) {
        this.message = message
        this.author= author
    }

    getMessage() : string {
        return this.message+" Tweet from Author: "+this.author
    }
}
```

- Map example codes to the Observer Pattern template




```
import Tweet from "../module/Tweet";

export default interface IObserver {
    onTweet(tweet : Tweet): string
}
import Tweet from "../module/Tweet";

export default interface IObservable{

    sendTweet(tweet : Tweet): any
}
```




```
import IObserver from '../interface/IObserver'
import Author from './Author'
import Tweet from './Tweet'

export default class Follower implements IObserver {

    name : string

    constructor(name: string){
        this.name = name
    }

    onTweet(tweet: Tweet) {
        console.log( this.name+" you got tweet =>"+tweet.getMessage())
        return this.name+" you got tweet =>"+tweet.getMessage()
    }

}
```

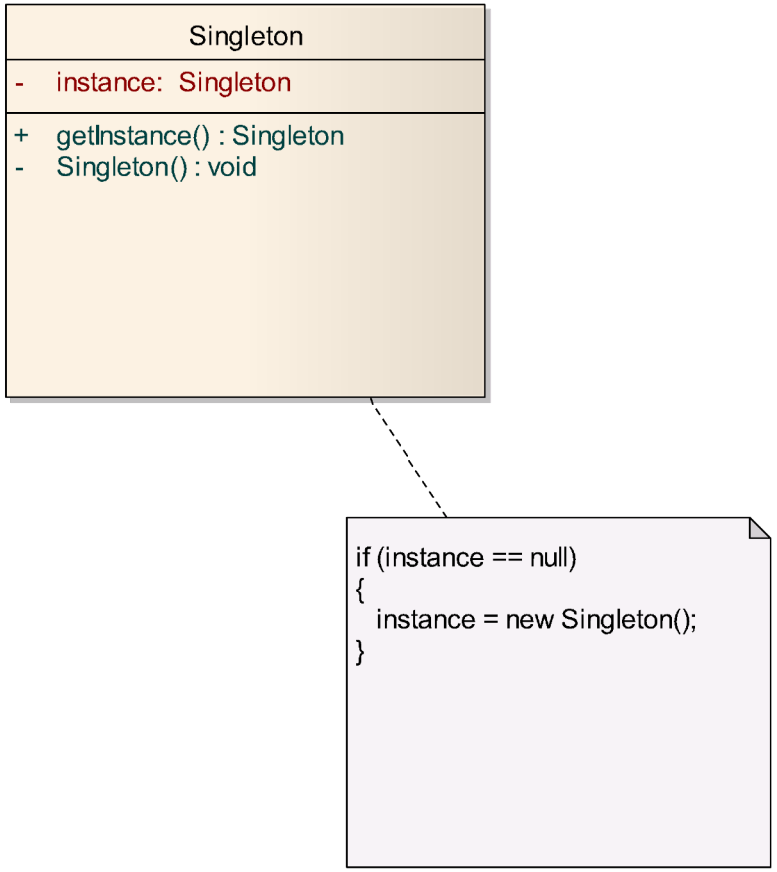
```
import IObservable from "../interface/IObservable";
import Tweet from "./Tweet";
import Follower from './Follower'
export default class Author implements IObservable {

    protected observers : Follower[] = []

    notify(tweet : Tweet){
        this.observers.forEach(observer => {
            observer.onTweet(tweet)
        })
    }

    subscribe(observer : Follower){
        this.observers.push(observer)
    }
    sendTweet(tweet : Tweet) {
        this.notify(tweet)
    }
}
```

# “Index.ts”

```ts
import express,{  Application, Request, Response } from 'express'
// import DBInstance from './helper/DB'
import bodyParser from 'body-parser'
import Follower from './module/Follower'
import Author from './module/Author'
import Tweet from './module/Tweet'

const app = express()

async function start(){

    try{

        app.use(bodyParser.json())
        app.use(bodyParser.urlencoded({extended : true}))
        // const db = await DBInstance.getInstance()

        app.post('/activate',async (req : Request,res : Response) => {
            try {

                const follower1 = new Follower("Ganesh")
                const follower2 = new Follower("Doe")

                const author = new Author()
                author.subscribe(follower1)
                author.subscribe(follower2)

                author.sendTweet(
                   new Tweet("Welcome","Bruce Lee")
                )

                res.status(200).json({ success : true,data:null })

            }
            catch(e){
                console.log(e)
                res.status(500).json({ success : false,data : null })
            }
        })

        app.listen(4000,() => {
            console.log("Server is running on PORT 4000")
        })
    }
    catch(e){
        console.log("Error while starting the server",e)
    }
}

start()
```

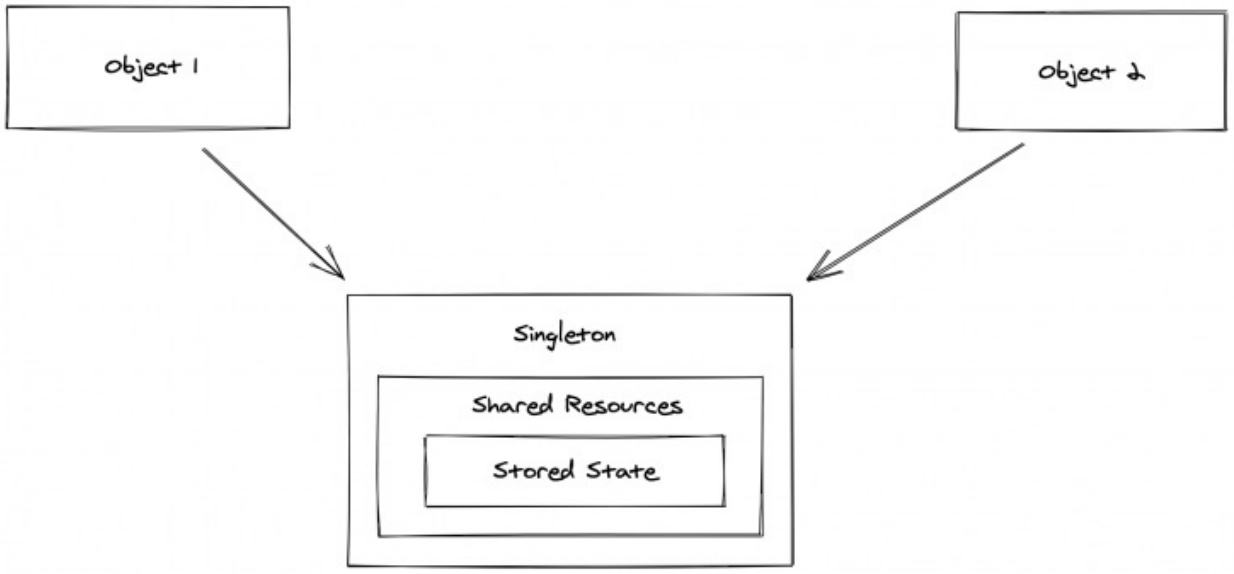
# 3. Singleton Pattern

- The singleton pattern implies that there should be only one instance for a class.
- By following this pattern, we can avoid having multiple instances for a particular class.



# Singleton Pattern – Example

- A good example of the Singleton pattern is database connection in our application.
- Having multiple instances of a database in our application makes an application unstable.
- So, the singleton pattern provides a solution to this problem by managing a single instance across the application.

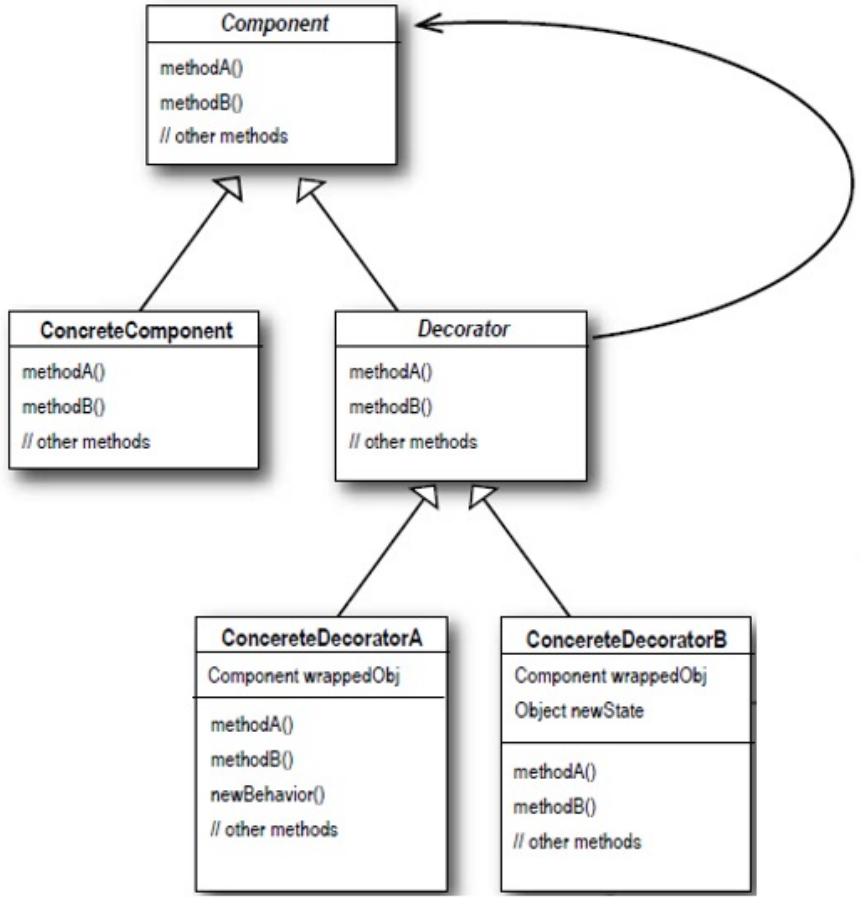


```
import {MongoClient,Db} from 'mongodb'
class DBInstance {

    private static instance: Db

    private constructor(){}

    static getInstance() {
        if(!this.instance){
            const URL = "mongodb://localhost:27017"
            const dbName = "sample"
            MongoClient.connect(URL,(err,client) => {
                if (err) console.log("DB Error",err)
                const db = client.db(dbName);
                this.instance = db
            })

        }
        return this.instance
    }
}

export default DBInstance
```

```
import express,{  Application, Request, Response } from 'express'
import DBInstance from './helper/DB'
import bodyParser from 'body-parser'
const app = express()

async function start(){

    try{

        app.use(bodyParser.json())
        app.use(bodyParser.urlencoded({extended : true}))
        const db = await DBInstance.getInstance()

        app.get('/todos',async (req : Request,res : Response) => {
            try {
                const db = await DBInstance.getInstance()

                const todos = await db.collection('todo').find({}).toArray(
                res.status(200).json({success : true,data : todos})
            }
            catch(e){
                console.log("Error on fetching",e)
                res.status(500).json({ success : false,data : null })
            }

        })

        app.post('/todo',async (req : Request,res : Response) => {
            try {
                const db = await DBInstance.getInstance()

                const todo = req.body.todos
               const todoCollection =  await db.collection('todo').insertOn
                res.status(200).json({ success : true,data : todoCollection
            }
            catch(e){
                console.log("Error on Inserting",e)
                res.status(500).json({ success : false,data : null })
            }
        })

        app.listen(4000,() => {
            console.log("Server is running on PORT 4000")
        })
    }
    catch(e){
        console.log("Error while starting the server",e)
    }
}

start()
```

# 4. Decorator Pattern

- There is a base class with methods and properties.
- Then, there are some instances of the class that need methods or properties that didn't come from the base class.

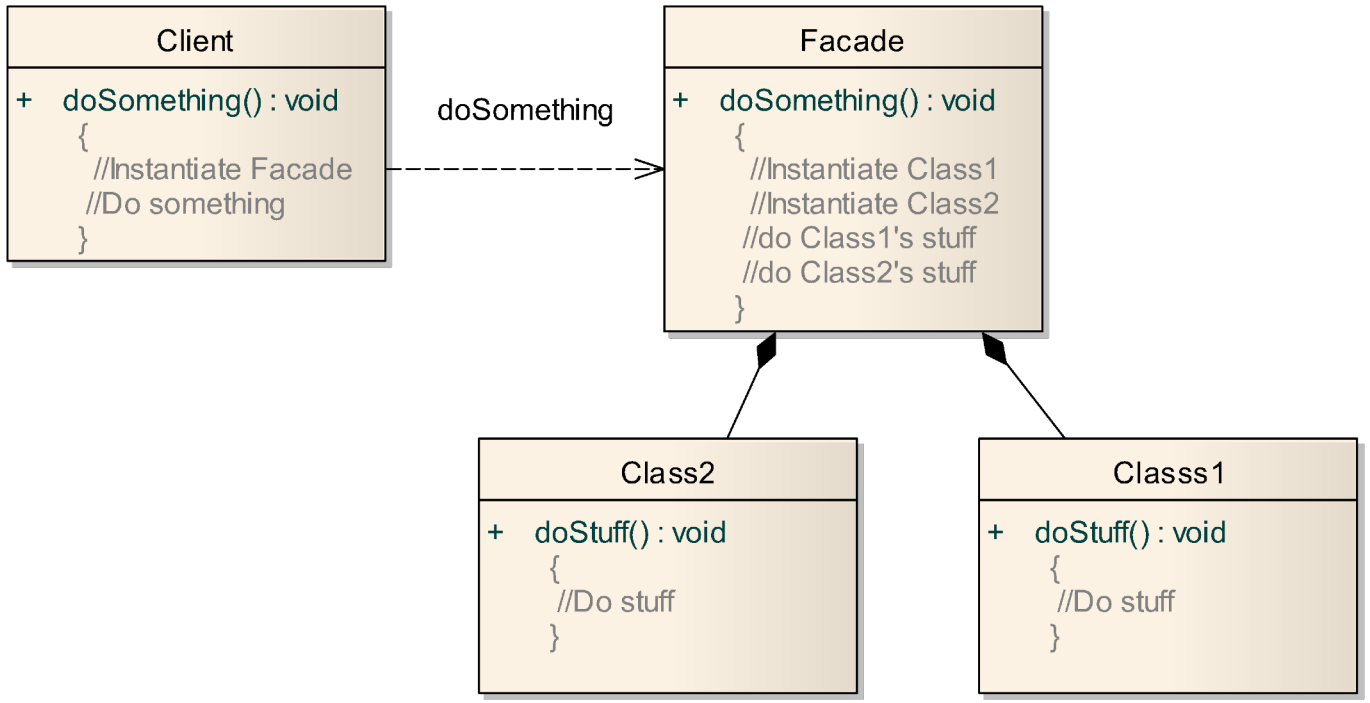


# Decorator Pattern – Example

```
class Sandwich {
  constructor(type, price) {
    this.type = type
    this.price = price
  }

  order() {
    console.log(`You ordered a ${this.type} sandwich for $ ${this.price}.`)
  }
}

class DeluxeSandwich {
  constructor(baseSandwich) {
    this.type = `Deluxe ${baseSandwich.type}`
    this.price = baseSandwich.price + 1.75
  }
}

class ExquisiteSandwich {
  constructor(baseSandwich) {
    this.type = `Exquisite ${baseSandwich.type}`
    this.price = baseSandwich.price + 10.75
  }

  order() {
    console.log(`You ordered an ${this.type} sandwich. It's got everything you need
  }
}

module.exports = { Sandwich, DeluxeSandwich, ExquisiteSandwich }
```

# 5. Facade Pattern

- The facade pattern allows us to wrap similar functions or modules inside a single interface.
- That way, the client doesn't need to know anything about how it works internally.

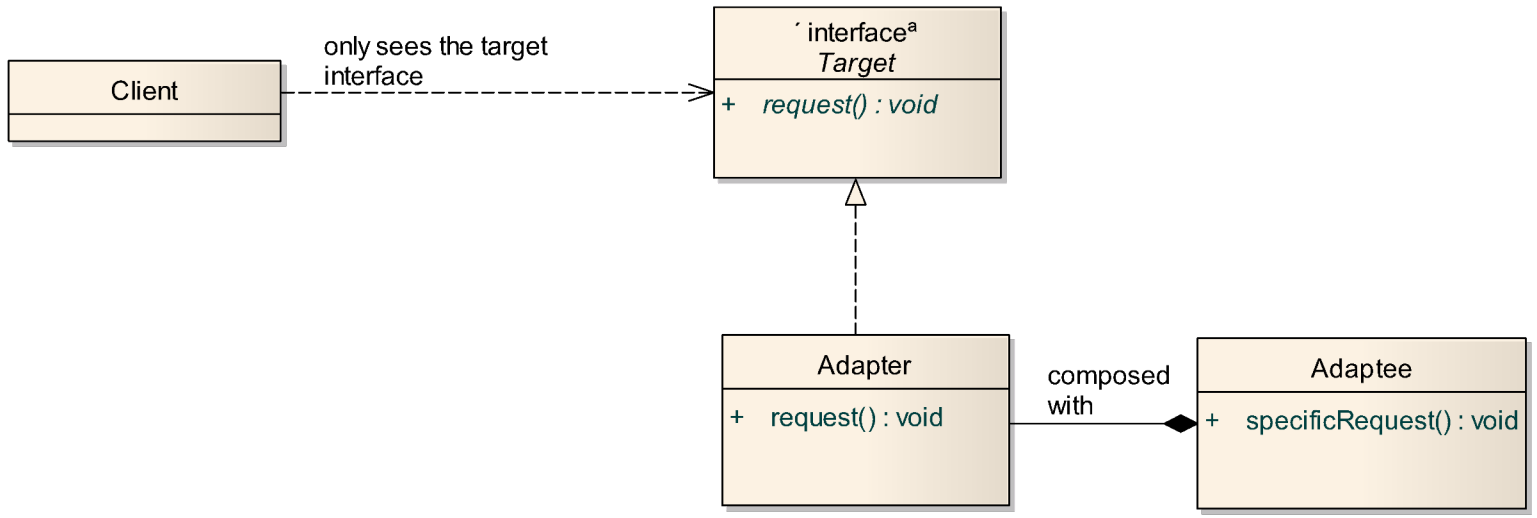


```ts
import IUser from '../Interfaces/IUser'

export default class User {
    private firstName: string
    private lastName: string
    private bankDetails: string | null
    private age: number
    private role: string
    private isActive: boolean

    constructor({firstName,lastName,bankDetails,age,role,isActive} : IUser) {
        this.firstName = firstName
        this.lastName = lastName
        this.bankDetails = bankDetails
        this.age = age
        this.role = role
        this.isActive = isActive
    }
    getBasicInfo() {
        return {
            firstName: this.firstName,
            lastName: this.lastName,
            age : this.age,
            role: this.role
        }
    }

    activateUser() {
        this.isActive = true
    }

    updateBankDetails(bankInfo: string | null) {
        this.bankDetails= bankInfo
    }

    getBankDetails(){
        return this.bankDetails
    }

    deactivateUser() {
        this.isActive = false
    }

    getAllDetails() {
        return this
    }
}
export default interface IUser {
    firstName: string
    lastName: string
    bankDetails: string
    age: number
    role: string
    isActive: boolean
}
```

```
import User from '../module/User'

export default class UserFacade{

    protected user: User
    constructor(user : User){
        this.user = user
    }

    activateUserAccount(bankInfo : string){
        this.user.activateUser()
        this.user.updateBankDetails(bankInfo)

        return this.user.getAllDetails()
    }

    deactivateUserAccount(){
        this.user.deactivateUser()
        this.user.updateBankDetails(null)
    }

}
```

# 6. Adaptor Pattern

- A classic example of an adapter pattern will be a differently shaped power socket.
- Sometimes, the socket and device plug doesn't fit.
- To make sure it works, we will use an adapter.



```
import IError from '../interface/IError'
export default class CustomError implements IError{

    message : string

    constructor(message : string){
        this.message = message
    }

    serialize() {
        return this.message
    }
}
```

```
export default class NewCustomError{

    message : string

    constructor(message : string){
        this.message = message
    }

    withInfo() {
        return { message : this.message }
    }
}
```

```
import NewCustomError from './NewCustomError'
// import CustomError from './CustomError'
export default class ErrorAdapter {
    message : string;
    constructor(message : string) {
        this.message = message
    }

    serialize() {
                // In future replace this function
        const e = new NewCustomError(this.message).withInfo()
        return e
    }

}
```

# For more details

- https://www.freecodecamp.org/news/4-design-patterns-to-use-in-web-development/
- https://blog.logrocket.com/design-patterns-in-typescript-and-node-js/