3 Принципи проектування класів SOLID

3.1 Огляд проблем, що зустрічаються при проектуванні і розробці класів

Якщо клас відповідає за кілька операцій відразу, ймовірність виникнення багів зростає – вносячи зміни, які стосуються однієї з операцій, ви, самі того не підозрюючи, можете торкнутися й інших.

Коли ви змінюєте поточну поведінку класу, ці зміни позначаються на всіх системах, які працюють із цим класом. Якщо хочете, щоб клас виконував більше операцій, то ідеальний варіант – не замінювати старі на нові, а додавати нові до існуючих.

У випадках, коли клас-нащадок не здатний виконувати ті ж дії, що і клас-батько, виникає ризик появи помилок.

Якщо у вас є клас і ви створюєте на його базі інший клас, вихідний клас стає батьком, а новий його нащадком. Клас-нащадок повинен виконувати такі ж операції, як і клас-батько. Це називається спадковістю.

Необхідно, щоб клас-нащадок був здатний обробляти самі запити, як і батько, і видавати той самий результат. Або результат може відрізнятися, але при цьому ставитися до того ж типу. На малюнку це показано так: клас-батько подає каву (у будь-яких видах), отже, для класу-нащадка прийнятно подавати капучіно (різновид кави), але неприйнятно подавати воду.

Якщо клас-нащадок не задовольняє цим вимогам, значить, він дуже відрізняється від батька і порушує принцип.

Коли класу доводиться робити дії, які не несуть ніякої реальної користі, це виливається в марну витрату ресурсу, а у разі, якщо клас виконувати ці дії не здатний, веде до виникнення багів.

Клас повинен проводити лише ті операції, які необхідні для здійснення його функцій. Всі інші дії слід або видалити зовсім, або перемістити, якщо є можливість, що вони знадобляться іншому класу в майбутньому.

Модулі (або класи) верхнього рівня = класи, які виконують операцію за допомогою інструмента

Модулі (або класи) нижнього рівня = інструменти, які потрібні для виконання операцій

Абстракції – представляють інтерфейс, що з'єднує два класи

Деталі = специфічні характеристики роботи інструменту

Клас не повинен з'єднуватися з інструментом, який застосовує для виконання операції. Натомість він повинен бути з'єднаний з інтерфейсом, який допоможе встановити зв'язок між інструментом та класом.

Крім того, ні інтерфейс, ні клас, не повинні вникати в специфіку роботи інструменту. Навпаки, цей інструмент повинен відповідати вимогам інтерфейсу.

Термін “SOLID” являє собою акронім для набору практик проектування програмного коду та побудови гнучкої та адаптивної програми. Цей термін був введений відомим американським фахівцем у галузі програмування Робертом Мартіном (Robert Martin), більш відомим як “дядечко Боб” або Uncle Bob.

Сам акронім утворений за першими буквами назв SOLID-принципів:

1. Single Responsibility Principle (Принцип єдиного обов’язку)
2. Open/Closed Principle (Принцип відкритості/закритості)
3. Liskov Substitution Principle (Принцип підстановки Лісков)
4. Interface Segregation Principle (Принцип розподілу інтерфейсів)
5. Dependency Inversion Principle (Принцип інверсії залежностей)

Принципи SOLID – це не патерни, їх не можна назвати певними догмами, які треба обов’язково застосовувати при розробці, проте їх використання дозволить поліпшити код програми, спростити можливі його зміни та підтримку.

3.2 Приклади використання принципів SOLID

3.2.1 Принцип єдиного обов’язку (Single Responsibility Principle)

Кожен клас повинен виконувати лише один обов’язок. Це не означає, що в нього має бути тільки один метод. Це означає, що всі методи класу мають бути сфокусовані на виконання одного спільного завдання. Якщо є методи, які не відповідають меті існування класу, їх треба винести за його межі.

Наприклад, клас User. Його обов’язок надавати інформацію про користувача: ім’я, email і тип підписки, яку він використовує в сервісі.

```
enum SubscriptionTypes {
  BASIC = 'BASIC',
  PREMIUM = 'PREMIUM'
}

class User {
  constructor (
    public readonly firstName: string,
    public readonly lastName: string,
    public readonly email: string,
    public readonly subscriptionType: SubscriptionTypes,
    public readonly subscriptionExpirationDate: Date
  ) {}
```

```
  public get name(): string {
    return `${this.firstName} ${this.lastName}`;
  }

  public hasUnlimitedContentAccess() {
    const now = new Date();

    return this.subscriptionType === SubscriptionTypes.PREMIUM
      && this.subscriptionExpirationDate > now;
  }
}
```

Розглянемо метод hasUnlimitedContentAccess. На основі типу підписки він визначає, чи є у користувача необмежений доступ до контенту. Але ж стоп, хіба це відповідальність класу User робити такий висновок? Виходить, у класу User є дві мети для існування: надавати інформацію про користувача і робити висновок, який у нього доступ до контенту на основі підписки. Це порушує принцип Single Responsibility.

Чому існування методу hasUnlimitedContentAccess у класі User має негативні наслідки? Бо контроль над типом підписки розпливається по всій програмі. Крім класу User, можуть бути класи MediaLibrary та Player, які теж вирішуватимуть, що їм робити на основі цих даних. Кожен клас трактує по-своєму, що означає тип підписки. Якщо правила наявних підписок змінюються, треба оновлювати всі класи, оскільки кожен вибудував свій набір правил роботи з ними.

Видалимо метод hasUnlimitedContentAccess у класі User і створимо новий клас, який буде відповідати за роботу з підписками.

```
class AccessManager {
  public static hasUnlimitedContentAccess(user: User) {
    const now = new Date();

    return user.subscriptionType === SubscriptionTypes.PREMIUM
      && user.subscriptionExpirationDate > now;
  }

  public static getBasicContent(movies: Movie[]) {
    return movies.filter(movie => movie.subscriptionType === SubscriptionTypes.BASIC);
  }

  public static getPremiumContent(movies: Movie[]) {
    return movies.filter(movie => movie.subscriptionType === SubscriptionTypes.PREMIUM);
  }

  public static getContentForUserWithBasicAccess(movies: Movie[]) {
    return AccessManager.getBasicContent(movies);
  }

  public static getContentForUserWithUnlimitedAccess(movies: Movie[]) {
```

```
    return movies;
  }
}
```

Ми інкапсулювали всі правила роботи з підписками в одному класі. Якщо будуть зміни у правилах, вони залишаться тільки у цьому класі та не зачіплять інші.

Single Responsibility Principle стосується не тільки рівня класів — модулі класів теж потрібно проєктувати таким чином, щоб вони були вузько спеціалізовані.

Крім SOLID існує ще інший набір принципів проектування програмного забезепення — GRASP. Деякі його принципи перетинаються з SOLID. Якщо говорити про Single Responsibility Principle, то з GRASP можно співставити:

- інформаційний експерт (Information Expert) — об’єкт, який володіє повною інформацією з предметної області;
- низька зв’язаність (Low Coupling) і високе зчеплення (High Cohesion) — компоненти різних класів або модулів повинні мати слабкі зв’язки між собою, але компоненти одного класу або модуля мають бути логічно пов’язані або тісно взаємодіяти один з одним.

3.2.2 Принцип відкритості/закритості (Open/Close Principle)

Класи мають бути відкриті до розширення, але закриті для змін. Якщо є клас, функціонал якого передбачає чимало розгалужень або багато послідовних кроків, і є великий потенціал, що їх кількість буде збільшуватись, то потрібно спроєктувати клас таким чином, щоб нові розгалуження або кроки не призводили до його модифікації.

Напевно, кожен з нас бачив нескінченні ланцюжки if then else або switch. Щойно додається чергова умова, ми пишемо черговий if then else, змінюючи при цьому сам клас. Або клас виконує процес з багатьма послідовними кроками — і кожен новий крок призводить до його зміни. А це порушує Open/Close Principle.

Як можна розширювати клас і водночас не змінювати його? Розглянемо кілька способів.

```
class Rect {
  constructor(
    public readonly width: number,
    public readonly height: number
  ) { }
}

class Square {
  constructor(
```

```
    public readonly width: number
  ) { }
}

class Circle {
  constructor(
    public readonly r: number
  ) { }
}

class ShapeManager {
  public static getMinArea(shapes: (Rect | Square | Circle)[]): number {
    const areas = shapes.map(shape => {
      if (shape instanceof Rect) {
        return shape.width * shape.height;
      }

      if (shape instanceof Square) {
        return Math.pow(shape.width, 2);
      }

      if (shape instanceof Circle) {
        return Math.PI * Math.pow(shape.r, 2);
      }

      throw new Error('Is not implemented');
    });

    return Math.min(...areas);
  }
}
```

Як бачимо, додавання нових фігур буде призводити до модифікації класу ShapeManager. Оскільки площа фігури тісно пов'язана із самою фігурою, можна змусити фігури самостійно рахувати свою площу, привести їх до одного інтерфейсу, а тоді передавати їх у метод getMinArea.

```
interface IShape {
  getArea(): number;
}

class Rect implements IShape {
  constructor(
    public readonly width: number,
    public readonly height: number
  ) { }

  getArea(): number {
    return this.width * this.height;
  }
}

class Square implements IShape {
  constructor(
    public readonly width: number
```

```
  ) { }

  getArea(): number {
    return Math.pow(this.width, 2);
  }
}

class Circle implements IShape {
  constructor(
    public readonly r: number
  ) { }

  getArea(): number {
    return Math.PI * Math.pow(this.r, 2);
  }
}

class ShapeManager {
  public static getMinArea(shapes: IShape[]): number {
    const areas = shapes.map(shape => shape.getArea());
    return Math.min(...areas);
  }
}
```

Тепер, якщо у нас з'являться нові фігури, все, що потрібно зробити, — це імплементувати інтерфейс IShape. І клас ShapeManager відразу буде її підтримувати без жодних модифікацій.

А що робити, якщо не можемо додавати методи до фігур? Існують методи, які суперечать Single Responsibility Principle. Тоді можна скористатися шаблоном проєктування «Стратегія» (Strategy): створити множину схожих алгоритмів і викликати їх за певним ключем.

```
interface IShapeAreaStrategiesMap {
  [shapeClassName: string]: (shape: IShape) => number;
}

class ShapeManager {
  constructor(
    private readonly strategies: IShapeAreaStrategiesMap
  ) {}

  public getMinArea(shapes: IShape[]): number {
    const areas = shapes.map(shape => {

      const className = shape.constructor.name;
      const strategy = this.strategies[className];

      if (strategy) {
        return strategy(shape);
      }

      throw new Error(`Could not find Strategy for '${className}'`);
```

```
    });

    return Math.min(...areas);
  }
}

// Strategy Design Pattern
const strategies: IShapeAreaStrategiesMap = {
  [Rect.name]: (shape: Rect) => shape.width * shape.height,
  [Square.name]: (shape: Square) => Math.pow(shape.width, 2),
  [Circle.name]: (shape: Circle) => Math.PI * Math.pow(shape.r, 2)
};

const shapes = [
  new Rect(1, 2),
  new Square(1),
  new Circle(1),
];

const shapeManager = new ShapeManager(strategies);
console.log(shapeManager.getMinArea(shapes));
```

Перевага Strategy в тому, що є змога змінювати в рантаймі набір стратегій і спосіб їх вибору. Можна прочитати файл конфігурацій (.json, .xml, .yml) і на його основі збудувати стратегії. Тоді, якщо відбувається зміна стратегій, не потрібно розробляти нову версію програми і деплоїти її на сервери, достатньо підмінити файл з конфігураціями і сказати програмі, щоб та його знову прочитала. Крім того, стратегії можна реєструвати в Inversion of Control контейнері. У такому разі клас, який їх потребує, отримає стратегії автоматично на етапі створення.

Розглянемо тепер ситуацію, коли триває послідовна багатокрокова обробка даних. Якщо кількість кроків зміниться, нам доведеться змінювати клас.

```
class ImageProcessor {
...
  public processImage(bitmap: ImageBitmap): ImageBitmap {
    this.fixColorBalance(bitmap);
    this.increaseContrast(bitmap);
    this.fixSkew(bitmap);
    this.highlightLetters(bitmap);

    return bitmap;
  }
}
```

Застосуємо дизайн-патерн «Конвеєр» (Pipeline).

```
type PipeMethod = (bitmap: ImageBitmap) => void;

// Pipeline Design Pattern
class Pipeline {
  constructor(
    private readonly bitmap: ImageBitmap
  ) { }
```

```
  public pipe(method: PipeMethod) {
    method(this.bitmap);
  }

  public getResult() {
    return this.bitmap;
  }
}

class ImageProcessor {
  public static processImage(bitmap: ImageBitmap, pipeMethods: PipeMethod[]): ImageBitmap
 {
    const pipeline = new Pipeline(bitmap);
    pipeMethods.forEach(method => pipeline.pipe(method))

    return pipeline.getResult();
  }
}

const pipeMethods = [
  fixColorBalance,
  increaseContrast,
  fixSkew,
  highlightLetters
];

const result = ImageProcessor.processImage(scannedImage, pipeMethods);
```

Тепер, якщо потрібно змінити спосіб обробки зображення, ми модифікуємо масив з методами. Сам клас ImageProcessor залишається незмінним. Тепер уявіть, що треба обробляти різні зображення по-різному. Замість того, щоб писати різні версії ImageProcessor, по-іншому скомбінуємо в масиві pipeMethods потрібні нам методи.

Ще кілька переваг. Раніше ми додавали новий метод обробки зображення прямо в ImageProcessor, і в нас виникала потреба додавати нові залежності. Наприклад, метод highlightLetters вимагає додаткову бібліотеку для пошуку символів на зображенні. Відповідно, більше методів — більше залежностей. Зараз кожен PipeMethod можна розробити в окремому модулі й підключати тільки необхідні залежності. Після такої декомпозиції все дуже легко тестувати. Ну й на останок: така структура коду мотивує розробника писати якомога коротші методи обробки з чіткими інтерфейсами. До цього можна було зробити один великий метод fixQuality, де б відбувалося і виправлення балансу кольорів, і вирівнювання зображення, і збільшення контрасту. Але в такому великому методі було б складно контролювати параметри кожного накладеного на зображення фільтру. Ймовірно, виникла б ситуація, коли fixQuality працював

би добре для одного зразка зображення, але для іншого на етапі тестування він би не працював зовсім. Маючи кілька добре програнульованих методів, значно простіше скоригувати параметри, щоб отримати потрібний результат.

Принципами GRASP, що спільні з Open/Close Principle :

- стійкість до змін (Protected Variations): потрібно захищати компоненти від впливу змін інших компонентів. Тому для компонентів, які потенційно часто будуть зазнавати змін, ми створюємо один інтерфейс і кілька його імплементацій, використовуючи поліморфізм;
- поліморфізм (Polymorphism) — можливість мати різні варіанти поведінки на основі типу класу. Типи класу з варіативною поведінкою мають підпадати під один інтерфейс;
- перенаправлення (Indirection): слабка зв'язаність між компонентами та можливість їх перевикористання досягається завдяки створенню посередника (mediator), який бере на себе взаємодію між компонентами.
- чиста вигадка (Pure Fabrication): можна створити штучний об'єкт, якого немає в домені, але який наділений властивостями, що дають змогу зменшити залежність між об'єктами. Наприклад, в домені є товар і склад. Якщо зробимо так, що склад буде контролювати наявність товарів, буде складно створити функціонал, який, наприклад, перевірятиме наявність товару в партнерів і пропонуватиме його користувачу. Тому ми додаємо об'єкт ProductManager, який перевірятиме, чи є товар на складі. Якщо немає, перевірятиме його в партнерів. Оскільки за допомогою ProductManager ми відв'язали товар від складу, можемо повністю позбутися його та продавати товари від партнерів, якщо виникне така потреба.

3.3 Принцип підстановки Лісков (Liskov Substitution Principle)

Якщо об'єкт базового класу замінити об'єктом його похідного класу, то програма має продовжувати працювати коректно.

Ящо ми перевизначаємо похідні методи від батьківського класу, то нова поведінка не має суперечити поведінці базового класу. Як приклад порушення цього принципу розглянемо такий код:

```
class BaseClass {
  public add(a: number, b: number): number {
    return a + b;
  }
}

class DerivedClass extends BaseClass {
```

```
  public add(a: number, b: number): number {
    throw new Error('This operation is not supported');
  }
}
```

Можливий також випадок, що батьківський метод буде суперечити логіці класів-нащадків. Розглянемо наступну ієрархію транспортних засобів:

```
class Vehicle {
  accelerate() {
    // implementation
  }

  slowDown() {
    // implementation
  }

  turn(angle: number) {
    // implementation
  }
}

class Car extends Vehicle {
}

class Bus extends Vehicle {
}
```

Все працює до того моменту, допоки ми не додамо новий клас Поїзд.

```
class Train extends Vehicle {
  turn(angle: number) {
    // is that possible?
  }
}
```

Оскільки поїзд не може змінювати довільно напрямок свого руху, то turn батьківського класу буде порушувати принцип підстановки Лісков.

Щоб виправити цю ситуацію, ми можемо додати два батьківські класи: FreeDirectionalVihicle — який буде дозволяти довільний напрямок руху і BidirectionalVehicle — рух тільки вперед і назад. Тепер всі класи будуть наслідувати тільки ті методи, які можуть забезпечити.

```
class FreeDirectionalVehicle extends Vehicle {
  turn(angle: number) {
    // implementation
  }
}

class BidirectionalVehicle extends Vehicle {
}
```

Крім того, клас-нащадок не має додавати ніяких умов до виконання методу і після виконання методу. Наприклад:

```
class Logger {
  log(text: string) {
```

```
    console.log(text);
  }
}

class FileLogger extends Logger {
  constructor(private readonly path: string) {
    super();
  }

  log(text: string) {
    // append text file
  }
}

class TcpLogger extends Logger {
  constructor(private readonly ip: string, private readonly port: number) {
    super();
  }

  log(text: string) {
    // implementation
  }

  openConnection() {
    // implementation
  }

  closeConnection() {
    // implementation
  }
}
```

У цій ієрархії ми не зможемо легко замінити об'єкти батьківського класу FileLogger об'єктами TcpLogger, оскільки до та після виклику методу нам потрібно додатково викликати openConnection та closeConnection. Виходить, ми накладаємо 2 додаткові умови на виклик методу log, що також порушує принцип підстановки Лісков.

Щоб вирішити ситуацію вище, ми можемо зробити методи openConnection та closeConnection приватними. В методі log класу TcpLogger організуємо запис логів в файл. Періодично (наприклад, кожної хвилини) відкриватимемо з'єднання, відправлятимемо файл з логами і закриватемо з'єднання. Додатково потрібно переконатися, що перед тим, як програма буде закрита, ми відправили всі логи. Якщо програма була аварійно завершена, ми можемо відправити логи під час наступного її запуску.

3.3.1 Принцип розділення інтерфейсу (Interface Segregation Principle)

Краще, коли є багато спеціалізованих інтерфейсів, ніж один загальний. Маючи один загальний інтерфейс, є ризик потрапити в ситуацію, коли похідний клас логічно не зможе успадкувати якийсь метод. Розглянемо приклад:

```
interface IDataSource {
  connect(): Promise<boolean>;
  read(): Promise<string>;
}

class DbSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }

  read(): Promise<string> {
    // implementation
  }
}

class FileSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }

  read(): Promise<string> {
    // implementation
  }
}
```

Оскільки з файлу ми читаємо локально, то метод Connect зайвий. Розділимо загальний інтерфейс IDataSource:

```
interface IDataSource {
  read(): Promise<string>;
}

interface IRemoteDataSource extends IDataSource {
  connect(): Promise<boolean>;
}

class DbSource implements IRemoteDataSource {
}

class FileSource implements IDataSource {
}
```

Тепер кожна імплементація використовує тільки той інтерфейс, який може забезпечити.

3.3.2 Принцип інверсії залежностей (Dependency Inversion Principle)

Він складається з двох тверджень:

- високорівневі модулі не повинні залежати від низькорівневих. І ті, і ті мають залежати від абстракцій;

- абстракції не мають залежати від деталей реалізації. Деталі реалізації повинні залежати від абстракцій.

Розберемо код, який порушує ці твердження:

```
class UserService {
  async getUser(): Promise<User> {
    const now = new Date();

    const item = localStorage.getItem('user');
    const cachedUserData = item && JSON.parse(item);

    if (cachedUserData && new Date(cachedUserData.expirationDate) > now) {
      return cachedUserData.user;
    }

    const response = await fetch('/user');
    const user = await response.json();

    const expirationDate = new Date();
    expirationDate.setHours(expirationDate.getHours() + 1);

    localStorage.setItem('user', JSON.stringify({
      user,
      expirationDate
    }));

    return user;
  }
}
```

Наш модуль верхнього рівня UserService використовує деталі реалізації трьох модулів нижнього рівня: localStorage, fetch та Date. Такий підхід поганий тим, що якщо ми, наприклад, вирішимо замість fetch користуватися бібліотекою, яка робить HTTP-запити, то доведеться переписувати UserService. Крім того, такий код важко покрити тестами.

Ще одним порушенням є те, що з методу getUser ми повертаємо реалізований клас User, а не його абстракцію — інтерфейс IUser.

Створимо абстракції, з якими було б зручно працювати всередині модуля UserService.

```
interface ICache {
  get<T>(key: string): T | null;
  set<T>(key: string, user: T): void;
}

interface IRemoteService {
  get<T>(url: string): Promise<T>;
}

class UserService {
  constructor(
    private readonly cache: ICache,
```

```
    private readonly remoteService: IRemoteService
  ) {}

  async getUser(): Promise<IUser> {
    const cachedUser = this.cache.get<IUser>('user');

    if (cachedUser) {
      return cachedUser;
    }

    const user = await this.remoteService.get<IUser>('/user');
    this.cache.set('user', user);

    return user;
  }
}
```

Як бачимо, код вийшов значно простішим і його можна легко протестувати. Тепер поглянемо на реалізацію інтерфейсів ICache та IRemoteService.

```
interface IStorage {
  getItem(key: string): any;
  setItem(key: string, value: string): void;
}

class LocalStorageCache implements ICache {
  private readonly storage: IStorage;

  constructor(
        getStorage = (): IStorage => localStorage,
        private readonly createDate = (dateStr?: string) => new Date(dateStr)
  ) {
        this.storage = getStorage()
  }

  get<T>(key: string): T | null {
        const item = this.storage.getItem(key);
        const cachedData = item && JSON.parse(item);

        if (cachedData) {
        const now = this.createDate();

        if (this.createDate(cachedData.expirationDate) > now) {
        return cachedData.value;
        }
        }

        return null;
  }

  set<T>(key: string, value: T): void {
        const expirationDate = this.createDate();
        expirationDate.setHours(expirationDate.getHours() + 1);

        this.storage.setItem(key, JSON.stringify({
        value,
```

```
            expirationDate
            }));
    }
}

class RemoteService implements IRemoteService {
    private readonly fetch: ((input: RequestInfo, init?: RequestInit) => Promise<Response>)

    constructor(
            getFetch = () => fetch
    ) {
            this.fetch = getFetch()
    }

    async get<T>(url: string): Promise<T> {
            const response = await this.fetch(url);
            const obj = await response.json();

            return obj;
    }
}
```

Ми зробили врапери над localStorage та fetch. Важливим моментом у реалізації двох класів є те, що ми не використовуємо localStorage та fetch прямо. Ми весь час працюємо зі створеними для них інтерфейсами. LocalStorage та fetch будуть передаватися в конструктор, якщо там не буде вказано жодних параметрів. Для тестів можна створити mocks або stubs, які замінять localStorage або fetch, і передати їх як параметри в конструктор.

Схожий прийом використовують і для дати: якщо нічого не передати, то кожного разу LocalStorageCache буде отримувати нову дату. Якщо ж для тестів потрібно зафіксувати певну дату, треба передати її в параметрі конструктора.

Висновки

Це природно, що з розвитком системи зростає її складність. Важливо завжди тримати цю складність під контролем. Інакше може виникнути ситуація, коли додавання нових фіч, навіть не дуже складних, обійдеться занадто дорого. Деякі проблеми повторюються особливо часто. Щоб їх уникати, було розроблено принципи проєктування. Якщо будемо їх дотримуватися, то не допустимо лавиноподібного підвищення складності системи. Найпростішими такими принципами є SOLID.