1 слайд

Принципи **SOLID** та як їх застосувати — те, що питають розробників на більшості співбесід. Це набір правил, який в теорії дозволяє створити відмовостійкий масштабований та легкий у підтримці продукт, код якого буде репрезентативним (характерний, типовий для чого-небудь). Водночас є низка сценаріїв, коли ці принципи не застосовуються та суперечать деяким шаблонам проєктування. Зараз я поділюся кейсами та прикладами, а також розповім про поширені помилки в інтерпретації цих правил.

2 слайд

Хто такий Роберт Мартін?

Роберт Сесіл Мартін, також відомий як **"Дядько Боб"** — програміст, автор і спікер, який здобув широку популярність завдяки своїм дослідженням у сфері розробки програмного забезпечення.

Він є автором багатьох книг та статей про програмування, зокрема **"Чистий код: Посібник з гнучкого розроблення програмного забезпечення"**, який вважається класичним посібником із написання чистого та підтримуваного коду.

Також він є творцем **SOLID принципів** — набору правил для розробки масштабованих і зручних для супроводу програмних систем.

Мартін активно популяризує **гнучкі методології розробки програмного забезпечення (Agile)** та є одним із співавторів **Agile Manifesto (**Маніфест гнучкої розробки — це документ, що описує основні принципи, на яких базується гнучка розробка).

Основні ідеї Agile:

- люди і взаємодія важливіше процесів та інструментів;

- працюючий продукт важливіше вичерпної документації;

- співпраця з замовником важливіше узгодження умов контракту;

- готовність до змін важливіше проходження початкового плану.

Мартін виступає на численних конференціях і семінарах по всьому світу, а також займається менторством і навчанням програмістів. Він - авторитетний експерт у сфері розробки програмного забезпечення, а його праці суттєво вплинули на підходи до програмування та супроводу коду.

3 слайд

Що таке SOLID?

Кожна літера у слові SOLID представляє один з принципів:

4 слайд

S — принцип єдиної відповідальності (Single Responsibility Principle, SRP). Введений Робертом Мартіном у книзі «Agile Software Development: Principles, Patterns, and Practices» 2002 року. Мартін підкреслив важливість створення класів, які відповідають лише за один напрям дій.

5 слайд

O — принцип відкритості/закритості (Open/Closed Principle, OCP). Вперше описаний Бертраном Меєром у книзі «Object-Oriented

Software Construction» 1988 року. Він визначив, що класи повинні бути відкритими для розширення, але закритими для модифікації, щоби сприяти гнучкості та легкості розширення коду.

6 слайд

L — принцип заміщення Лісков (Liskov Substitution Principle, LSP). Сформульований Барбарою Лісков 1988 року. Її робота «A Behavioral Notion of Subtyping» визначила умови, за якими один об'єкт може замінити інший без змін поведінки програми.

7 слайд

I — принцип розділення інтерфейсу (Interface Segregation Principle, ISP). Введений Робертом Мартіном, він розширив ідеї використання малих та специфічних інтерфейсів для зменшення залежностей.

8 слайд

D — принцип інверсії залежності (Dependency Inversion Principle, DIP). Сформульований Робертом Мартіном, визначає важливість того, щоби модулі низького рівня не залежали від верхніх модулів, а обидва типи залежали від абстракцій.

Набір принципів SOLID визначив основи для створення легко зрозумілих, підтримуваних та розширюваних систем.

9 слайд

Навіщо потрібні принципи SOLID?

SOLID розширює список базових принципів об'єктно-орієнтованого програмування. «Якщо ООП розповідає нам, що таке класи, і що вони можуть, то SOLID пояснює, яким чином краще їх поєднувати між собою, будувати та компонувати систему. Фактично цей набір принципів допомагає інтегрувати принципи ООП, підштовхує розробників до використання абстракцій та правильної побудови залежностей в системі. Набагато простіше працювати з кодом, що підготовлений до розширення і перевикористання загалом, — таким чином нам достатньо фокусуватися на власному коді замість того, щоби розпиляти увагу на вивчення коду, що існує».

Загалом SOLID націлений на якість коду. Він допомагає створювати код, який буде легко перевикористовувати, масштабувати та підтримувати. Гарним показником якості застосунку є те, наскільки просто і зрозуміло з ним працювати іншим розробникам. Хороший код сам себе репрезентує, пояснює деталі складових продукту, слугуючи не тільки інструкцією для машини, а ще й документом для розробника.

«Як саме мають бути реалізовані ці принципи, вирішує техлід або CTO (Chief technology officer) — це менеджер компанії, відповідальний за розробку нових послуг) проєкту, тому підходи до їхнього застосування різняться. Водночас незалежно від архітектури та інтерпретації, модуль, написаний по SOLID в одній системі, буде не сильно відрізнятися від модуля, написаного за цими принципами в іншій системі, адже обидва будуть наближені до однакової структури якості».

Розглянемо кожен принцип детальніше та розберемо приклади.

10 слайд

Принцип єдиної відповідальності (SRP)

Цей принцип стверджує, що кожен клас або модуль має бути відповідальним лише за одну конкретну задачу, отже, мати лише одну причину для зміни. Це допомагає зменшити залежність між різними частинами коду.

(розглядаємо ФОТО)

«Вивчаючи структуру великої системи, ми маємо обмежений контекст сприйняття і бачимо інформацію лише в певному радіусі. Якщо вся логіка буде описана в одному файлі, код займатиме десятки тисяч рядків, і ми нічого не зрозуміємо. Щоби ми могли сприймати систему, ми розбиваємо її на складові. Про це нам каже SRP. Цей принцип полягає у тому, щоби розділяти елементи системи за зоною їх призначення, щоби один модуль не мав більш ніж однієї зони відповідальності. У такий спосіб їх простіше вивчати, а також, чим менше відповідальностей має модуль, тим менше причин його змінювати, і тим менше потенційних причин його зламати.

Принцип SRP допомагає подолати поширений антипатерн GodObject (богоб'єкт), коли один об'єкт у системі виконує велику кількість функцій або має занадто велику кількість властивостей. Іноді його називають «Singleton (Одинак) на стероїдах» або «Монолітний об’єкт».

11 слайд

На прикладі на слайді — клас, що відповідає одразу за все. Відповідно, зміна в одній з частин класу поставить під загрозу всі інші.

12 слайд

В ідеалі розробники мають писати класи так, щоб їх не довелося часто змінювати. Отже, цей клас варто розбити на декілька. Відповідно тепер, якщо нам треба змінити логіку відправки імейла, це не вплине на інші функції.

У цього принципу є і зворотна сторона. Розробники можуть помилково вважати, що кожен клас повинен робити тільки одну функцію від самого початку. Це може призвести до надмірної грануляції, якщо не забезпечується баланс між атомізацією та зручністю в управлінні класами.

Натомість наслідування цього принципу полягає у двох напрямах. З одного боку варто розділяти великі класи, які містять багато функцій, а з іншого — уникати дрібних однотипних класів, розмазаних по коду, які важко об’єднувати за сенсом.

Декомпозуючи код, важливо балансувати, щоби кожен модуль був якомога струнκішим, але сприйняття системи залишалося зв’язним. Інколи ООП сприймається занадто буквально, і це перетворюється на підхід «розділю цей клас на ще три, бо так рекомендує SOLID». Такий код читати важко

13 слайд

Принцип відкритості/закритості (OCP)

Згідно з цим принципом, програмні сутності (класи, модулі, функції) повинні бути відкритими для розширення, але закритими для модифікації. Це допомагає зробити систему більш гнучкою,

забезпечити можливість додавати нову функціональність, не торкаючись старого коду. Наприклад, клас для обробки платежів має легко розширюватися для додавання нових методів оплати.

(розглядаємо ФОТО)

«Open/Closed — досить неоднозначний принцип, багато розробників неправильно інтерпретують його. Суть полягає у правильній побудові модулів, щоби розширення логіки не вимагало переписування старого коду»

14 слайд

Приклад порушення принципу Open/Closed.

У прикладі класу ClosedForExtensionChangesIsOnlyOptionToEdit, принцип Open/Closed порушений, оскільки для додавання нового функціоналу (наприклад, отримання статистики з іншого джерела чи виконання додаткової обробки даних) потрібно змінювати сам клас, а не розширювати його.

15 слайд

Рішення: Щоб дотримуватись принципу Open/Closed, можна зробити клас відкритим для розширення, наприклад, через використання інтерфейсів або абстракцій.

В цьому прикладі ви можете додавати нові реалізації `StatsProvider`, не змінюючи клас `ClosedForExtensionChangesIsOnlyOptionToEdit`, таким чином розширюючи функціонал, але не змінюючи вже існуючий код.

Тут вже створено інтерфейс `StatsProvider`, який визначає метод `getStats($userId)`. Це дозволяє різним класам, що реалізують цей інтерфейс, надавати свою власну реалізацію отримання статистики (наприклад, з бази даних, з API тощо).

Також клас DbStatsProvider, який реалізує інтерфейс StatsProvider і надає конкретну реалізацію для отримання статистики з бази даних (це той самий код, що був у початковому варіанті, але виноситься в окремий клас).

Замість того, щоб вбудовувати логіку отримання статистики безпосередньо у методі getStats, передається інтерфейс StatsProvider через конструктор класу. Це дозволяє класу бути відкритим для розширення (можна додавати нові типи постачальників статистики), але закритим для змін (не потрібно змінювати існуючий код класу, якщо ми хочемо змінити спосіб отримання статистики).

Як це працює?

Тепер, якщо потрібно додати новий спосіб отримання статистики (наприклад, з API), не потрібно змінювати код класу. Просто створюєте новий клас, що реалізує інтерфейс StatsProvider, і передаєте його в конструктор.

Клас ClosedForExtensionChangesIsOnlyOptionToEdit тепер є закритим для змін (він не потребує модифікацій, якщо змінюється спосіб отримання статистики), але відкритим для розширення (через додавання нових класів, що реалізують StatsProvider).

Таким чином, ми дотримуємось принципу Open/Closed.

Принцип заміщення Лісков (LSP)

Цей принцип каже, що підкласи базового класу повинні легко займати його місце без додаткових змін в коді. Це гарантує, що використовуючи поліморфізм під час внесення змін у систему, ми не «вистрілимо собі у ногу».

(розглядаємо ФОТО)

«Суть цього принципу не в тому, що ми можемо замінити батьківський клас дочірнім, а в тому, яким вимогам повинні відповідати дочірні класи. Якщо «батько» виконує функцію певним чином, то будь-яка «дитина» має зважати на це і не перекривати стару функціональність новою логікою. Наприклад, якщо у нас є базовий клас сповіщень. Він записує файл в системі, фіксує згадку в лог, що було відправлено повідомлення, а потім відправляє його, наприклад, на пошту. Якщо ми створюємо дочірній клас, в якому реалізується така ж функція відправки, але без логів. Виходить, що ми порушуємо принцип Лісков, адже підставляючи «дитину» на місце батька, ми втрачаємо логіку логування»

17 слайд

Наприклад, у нас є код:

```
class OldParent {

    public function notifySubscribers() {
        //log time
        //get subscribers
        //notify subscribers via email
    }
    protected function getSubscriberIds() {
        //giveIds
    }
}
```

Все було добре, але раптом нас попросили зробити кнопку, яка замість імейла відправить смс-повідомлення.

```
class NewChild extends OldParent {
    public function notifySubscribers() {
        //get subscribers from parent
        //notify subscribers via sms
    }
}
```

Через місяць до нас приходить продакт-менеджер і каже, що бюджет на відправку смс-повідомлень раптово витратився, і треба дізнатись, чому так сталося. Щоби визначитись, де все пішло не так, ми дивимося логи, але ми не бачимо жодної згадки про відправку смс. Здогадуєтеся, чому? Саме через порушення принципу Лісков: ми замінили стару реалізацію новою, забувши проконтролювати відповідність нової логіки до стандартів старої. Всі підтипи базового класу повинні бути взаємозамінними з «дітьми» без ризиків і необхідності поглиблення в деталі — про це і каже правило Лісков.

18 слайд

Правильний клас виглядав би так:

```
class NewChild extends OldParent {
    public function notifySubscribers() {
        //log time
        //get subscribers from parent
        //notify subscribers via sms
    }
}
```

Принцип Барбари Лісков зав’язаний на ідеї наслідування, від якої розробники намагаються відійти в останні роки, адже це передбачає прямі залежності від реалізації замість абстракцій. Тому це правило застосовується менше за інші.

19 слайд

Принцип розділення інтерфейсу (ISP)

«Залежність від багажу, який вам не потрібен, може спричинити проблеми, яких ви не очікували», — пише Роберт Мартін, описуючи цей принцип. Він вчить, що об’єкти не повинні залежати від методів інтерфейсів, які вони не використовують.

(розглядаємо ФОТО)

20 слайд

Приклад порушення ISP — створення одного великого інтерфейсу з багатьма методами, щоби передбачити всі можливі потреби. Клієнти, що залежать від цієї абстракції, можуть не потребувати її в повному обсязі, через що тягнутимуть зайвий багаж за собою. Щоби цей дизайн відповідав ISP, потрібно створити менші інтерфейси, які охоплюватимуть лише поведінку, яку використовує кожен із клієнтів.

21 слайд

Отже, замість одного загального інтерфейсу варто створювати декілька простих та мінімалістичних, які відображатимуть потреби клієнтів та не міститимуть зайвого функціоналу.

22 слайд

Принцип інверсії залежності (DIP)

Цей принцип каже, що низькорівневі модулі не повинні залежати від високорівневих, і обидва повинні залежати від абстракцій. Також

вказує на те, що абстракції не повинні залежати від деталей, а деталі мають залежати від абстракцій.

(розглядаємо ФОТО)

23 слайд

Розглянемо приклад порушення Dependency Inversion Principle

```
class IWorkWithThirdParty {}
class MysqlSomething {

    public function __construct(public IWorkWithThirdParty $view) {}
    public function checkData() {
        //get3party data
        //readDb
        //give result
    }

}

class Command {
    public function __construct(private MysqlSomething $mysqlSomething) {}
    public function __invoke() {
        return $this->mysqlSomething->getData();
    }
}
```

Тут порушено два ключові принципи: залежність від реалізації та залежність модуля MysqlSomething від сервісного рівня, що знаходиться вище і виступає прошарком між view і доменним рівнем.

24 слайд

Переробимо це відповідно до DIP:

```
interface PassCheck {
    public function passCheck();
}

interface GetIds {
    public function getIds();
}

class IWorkWithThirdParty implements GetIds {
    public function getIds() {
        //
    }
}
class MysqlSomething implements getIds {
    public function getIds() {
        //
    }
}
```

```
class ExternalDataConsistencyChecker implements PassCheck {

    public function __construct(private array $idSources) {}
    public function check() {
        $idVariations = [];

        foreach ($this->idSources as $idSource) {
            $idVariations[] = $idSource->getIds();
        }

        if (VARIATIONS_IS_NOT_EQUAL) {
            return false;
        }
        return true;
    }
}

class SyncExternalDataCommand {
    public function __construct(private PassCheck $idConsistencyCheck) {}
    public function __invoke() {
        if (!$idConsistencyCheck->passCheck()) {
            //pass check
        }
    }
}
```

Тепер всі модулі залежать тільки від абстракцій. Вся логіка, що не стосується бази, винесена нагору в сервісний рівень. Ієрархія залежностей від абстракцій вибудована коректно: згори вниз.

25 слайд

Як засвоїти принципи SOLID?

Початківці, які тільки починають вивчати принципи SOLID, можуть зіткнутися з низкою проблем.

Розуміння. Перша зустріч може бути складною, оскільки ці правила вимагають нового способу мислення щодо структури коду та взаємодії об'єктів.

Застосування. Навіть зрозумівши ці принципи у навчальних прикладах, може бути важко визначити, як саме застосовувати їх в конкретних робочих ситуаціях.

Внесення змін. Якщо значний обсяг кодової бази вже написаний без урахування SOLID, застосування цих принципів може викликати суперечності зі структурами, що існують.

Збільшення часу розробки. На етапі навчання та адаптації впровадження SOLID може спричинити складність коду або збільшення часу розробки. Проте у довгостроковій перспективі використання SOLID допомагає створити більш підтримуваний,

гнучкий та розширюваний код, зменшуючи загальний технічний борг.

Розробники повинні розуміти, що освоєння SOLID — це тривалий процес. Підвищення рівня навичок відбувається з часом. Поступове вивчення та постійна практика допомагають зрозуміти ці принципи та навчитися застосовувати в роботі. Вносячи зміни в існуючий код, важливо вміти адаптувати правила до реальних сценаріїв та змінювати код поетапно, не переписуючи його повністю.

Загалом не варто зациклюватися на застосуванні принципів. Вони не гарантують, що ваш код — хороший, і з ним зручно працювати. Початківцям краще концентруватися на загальних методиках якості коду, таких як відсутність повторення, простота умов if, обмежена кількість аргументів тощо. Якщо написати код з урахуванням загальних стандартів якості коду, ви несподівано відкриєте, що він цілком підпадає під SOLID

26 слайд

Список джерел

27 слайд

Q&A