

1. Single-responsibility principle

Класс сторедж може мати методи, що додають до нього продукти. Але якщо додати до нього метод виводу на екран, то це порушує принцип. Бо виводом на єкран ма займатись інший клас. Це не зона відповідальності стореджу

1. Open–closed principle

Маємо клас Нотіфікейшн. Він відправляє Смс методом Нотіфікейт. Заказчик хоче, щоб відправляв ще й Емейл. Ми не маємо перероблювати метод Нотіфікейт у цьому класі, додавати перевірки. Буде правильмин або зробити новий метод для цього, або ж створити клас-наслідник, де Оверрайднути метод, додавши до нього функціонал. Таким чином метод Нотіфікейт залишився закритим, незмінним та ми йоро розширили у насліднику.

1. Liskov substitution principle

Якщо у нас

1. Interface segregation principle

Ми робимо клас птаха. Він може рухатись та літати. Та створюем для цього інтерфейс АйПтахабл з методами польоту та ходьди. Теперь Створимо класс Тварини, що ходить. Тут буде АйМувабл інтерфейс із методом ходьби. Але якщо я захочу і того, і того засунути в лист ІМувабл - птаха я не зможу туди посадити, хоча він може ходити. Тому потрібно розбити інтерфейс АйПтахабл на АйМувабл (метод ходити) та АйФлаебл (метод польоту).

1. Dependency inversion principle

Замість того, щоб у конструктор передавати конкретний клас, можно передавати інтерфейс із контрактом, який має реалізовувати клас, що надходить у конструктор.

Візьмемо банк, що проводить оптаку кешем. Передаємо у нього клас Кеш, що робить транзакцію кешем. А тепер хочемо додавати оплату карткою. Опа, а що, а як?

А тепер виведемо з класу Кеш інтерфейс АйПеймент (метод Пей), та запихнемо його у конструктор. Кем реалізуватиме цей інтерфейс. Передали кеш у конструктор через інтерфейс - все топчик. А тепер додаємо оплату карткою. Робило клас Кард і реалізуемо інтерфейс. А тепер ьспокійно передаємо його у конструктор.   
Тепер завдяки цьому принципу наш банк не залежить від класу, що приходить до нбого, якщо він реаліхує інтерфейс