

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 екран 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 правильним 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

Ми повинні створювати спадкоємців будь-якого базового класу тоді й тільки тоді, коли вони збираються правильно реалізувати його логіку, не викликаючи проблем під час заміни батьків на спадкоємців.

Тобто створюючи класс тигра від классу ссавця та реалізуючи у нього метод ходьби, ми маємо оверрайдити цей метод так, щоб це не ламало логіку ходьби ссавців. Ми можемо додати до методу тихі подушки для безшумної та м’якої ходьби, але не маємо права додавати до методу елементи польоту (ті, хто ходять по морському дну, такого не оцінять).

1. Interface segregation principle

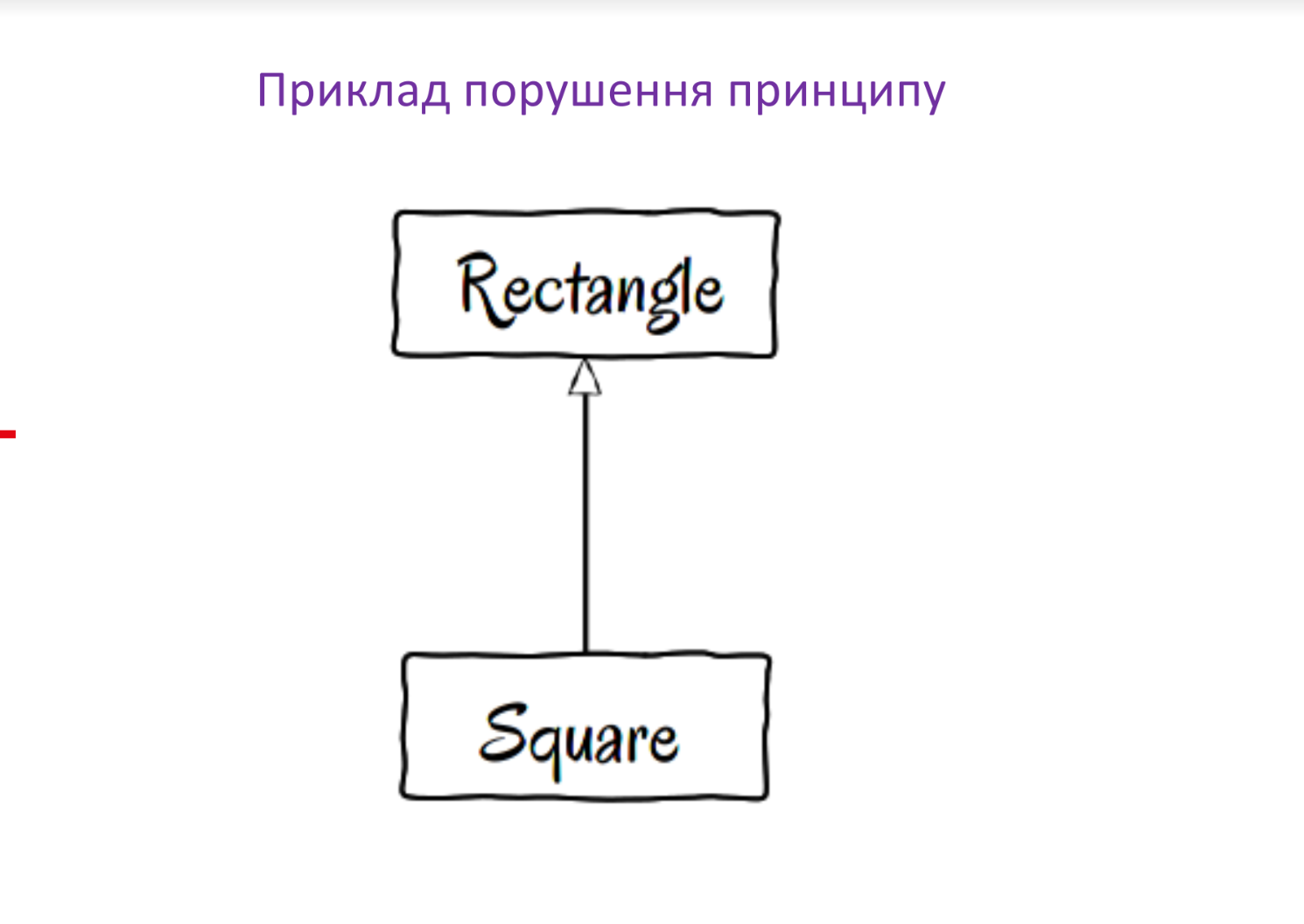
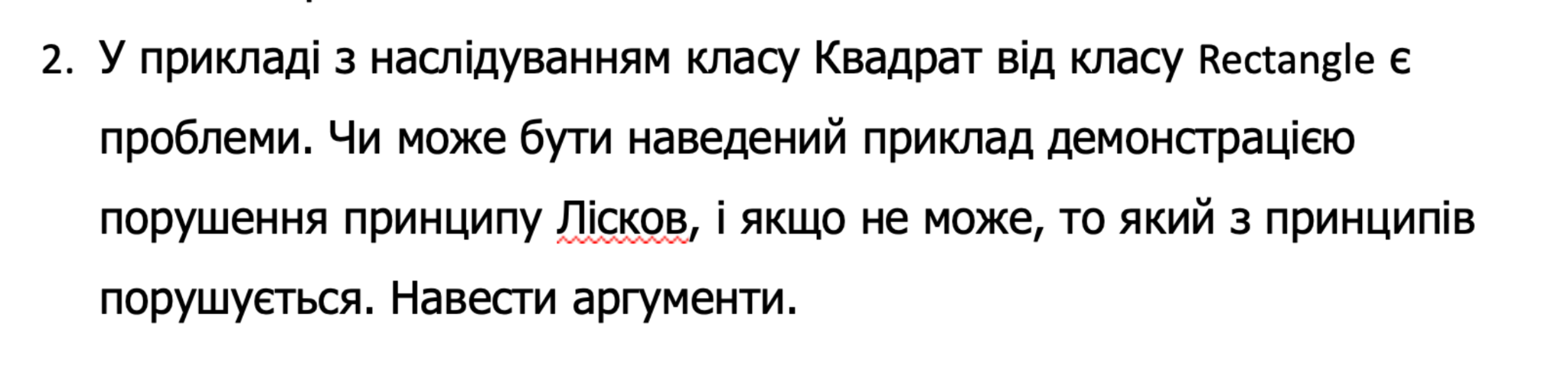
Ми робимо клас птаха. Він може рухатись та літати. Та створюємо для цього інтерфейс АйПтахабл з методами польоту та ходьби. Теперь Створимо класс Тварини, що ходить. Тут буде АйМувабл інтерфейс із методом ходьби. Але якщо я захочу і того, і того засунути в лист ІМувабл - птаха я не можу туди посадити, 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 оплату 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. Кеш реалізуватиме цей інтерфейс. Передали Кеш у конструктор через інтерфейс - все топчик. А тепер додаємо оплату карткою. Робимо клас Кард і реалізуємо інтерфейс. Спокійно передаємо його у конструктор.   
Тепер завдяки цьому принципу наш банк не залежить від класу, що приходить до нього, якщо він реалізує інтерфейс



Клас прямокутника має метод GetArea() та поля А та Б. Клас квадрат наслідується від класа прямокутника та (додумую) оверрайдить метод GetArea() так, щоб використовувати лише поле А (А\*А).

В данному прикладі заміна **у програмі, що використовує тип прямокутника**, тип прямокутника на квадрат, може спричинити його поломку через те, що програма розрахована на те, щоб А та Б могли бути різними. І в цьому випадку квадрат, що робить А\*А, просто забуває про поле Б, що призводить до неправильних обчислень.

Це і є порушенням принципу Барбари Лісков.

Як його пофіксити? Банально змінити напрям наслідування. Щоб клас ректенгл наслідувався від квадрату, залишаючи конструктор на лише сторону А (а = А, б = А), та мав поле Б. У такому випадку обчислення площі при підстановки замість квадрату прямокутника залишуться правильними і принцим підстановки Лісков не буде порушений.