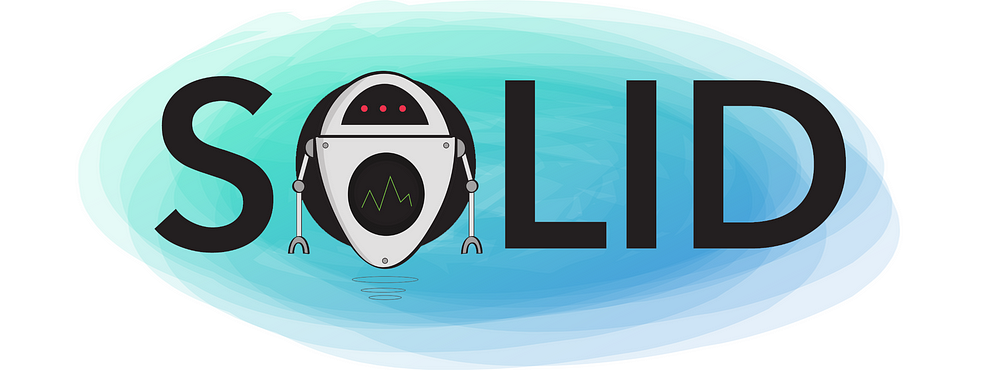
**Принципи SOLID у малюнках**



Якщо ви знайомі з [об’єктно-орієнтованим програмуванням](https://en.wikipedia.org/wiki/Object-oriented_programming) , то, мабуть, чули про принципи [**SOLID**](https://en.wikipedia.org/wiki/SOLID) .

Ці п’ять принципів розробки програмного забезпечення є рекомендаціями, яких слід дотримуватися під час створення програмного забезпечення, щоб його було легше масштабувати та підтримувати. Їх зробив популярним інженер-програміст [Роберт К. Мартін](https://en.wikipedia.org/wiki/Robert_C._Martin) .

В Інтернеті є так багато чудових статей про **SOLID** , але я рідко бачу приклади із зображеннями. Через це таким людям, які навчаються візуальному, як я, стає дещо складно вчитися, залишаючись залученими.

Отже, головна мета цієї статті — краще зрозуміти ці принципи, використовуючи ілюстрації та підкреслюючи мету кожного принципу.

Розумієте, деякі з цих принципів можуть виглядати схожими, але вони не спрямовані на **ту саму мету** . Можна задовольнити один принцип, порушуючи інший, навіть якщо вони схожі.

Щоб зробити це максимально простим, я буду використовувати слово « **Клас»** , але зауважте, що воно також може застосовуватися до **функції** , **методу** або **модуля** в цій статті.

***Оновлення\*****Я отримав кілька коментарів щодо відкритого закритого в цій статті порушенням принципу єдиної відповідальності. Зверніть увагу, що мета цієї статті — пояснити кожен із цих принципів незалежно від інших.*

*Крім того, обов’язки (або ролі) відрізняються від дій. У SRP я використовував «I am Painter», у Open-Closed я використовував «I can Paint».*

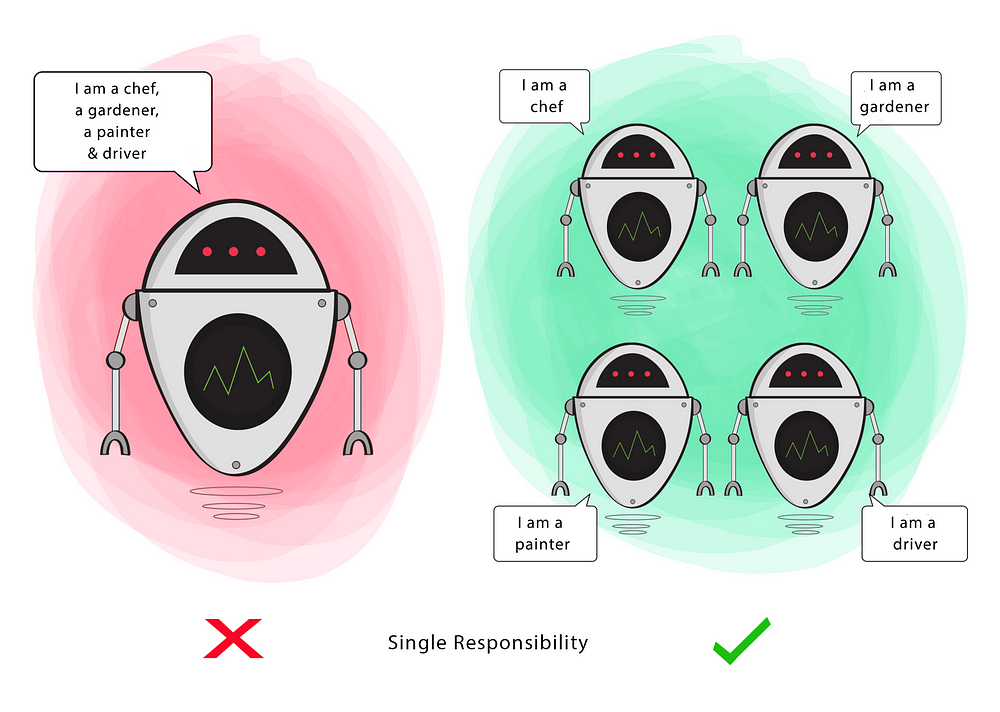
*Важливо звернути увагу на це, оскільки для виконання відповідальності (або ролі) можна виконати кілька дій. Клас повинен мати одну відповідальність (SRP), але його функціональні можливості, які виконують цю відповідальність, повинні бути відкритими для розширення (OCP).*

Тепер почнемо!

**Принципи SOLID**

**S — єдина відповідальність**

Клас повинен нести єдину відповідальність



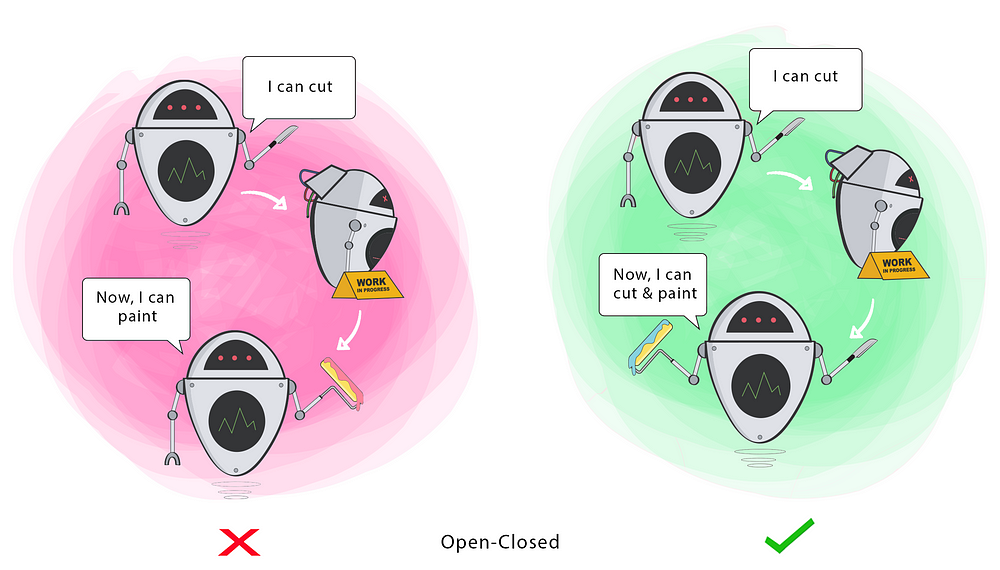
Якщо клас має багато обов’язків, це збільшує ймовірність помилок, оскільки внесення змін до одного з його обов’язків може вплинути на інші без вашого відома.

**Мета**

Цей принцип має на меті відокремити поведінку таким чином, що якщо в результаті ваших змін виникнуть помилки, це не вплине на іншу непов’язану поведінку.

**O — Відкрито-Закрито**

Класи мають бути відкритими для розширення, але закритими для змін



Зміна поточної поведінки класу вплине на всі системи, які використовують цей клас.

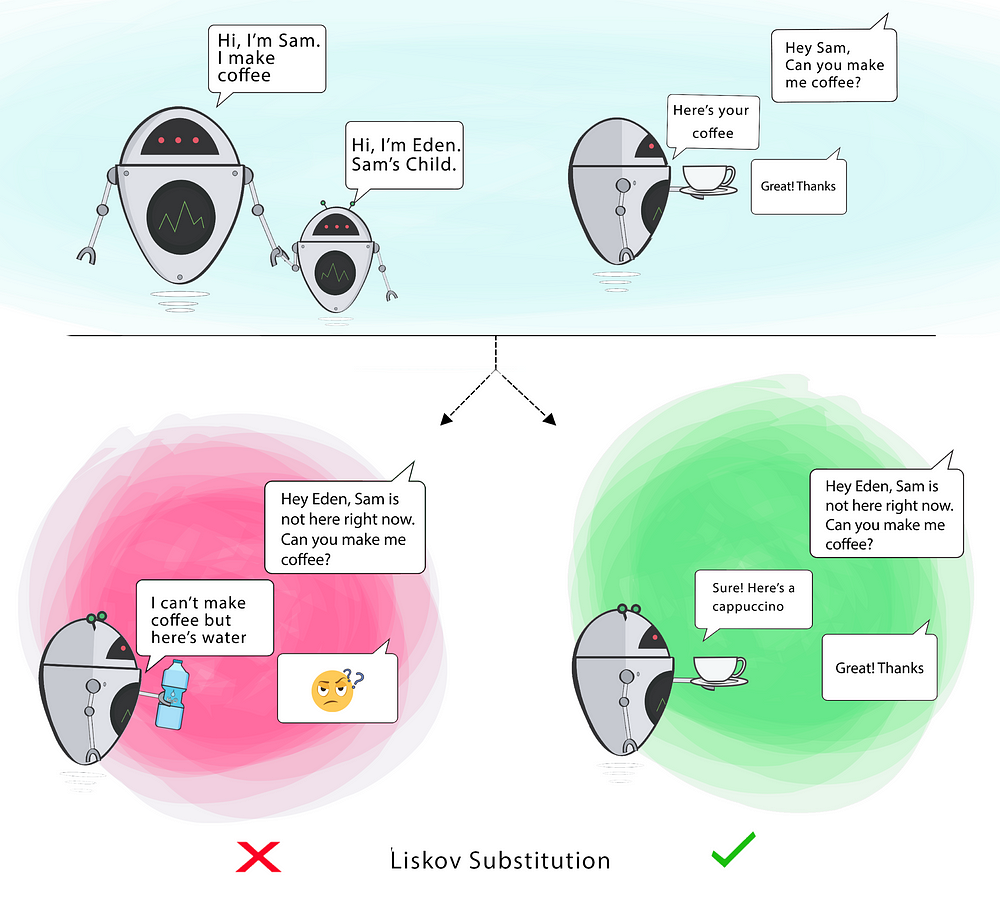
Якщо ви хочете, щоб клас виконував більше функцій, ідеальним підходом є додавання до вже існуючих функцій, НЕ змінюючи їх.

**Мета**

Цей принцип спрямований на розширення поведінки класу без зміни існуючої поведінки цього класу. Це робиться для того, щоб уникнути виникнення помилок у будь-якому місці, де використовується клас.

**L** — **Заміна Ліскова**

Якщо S є підтипом T, то об’єкти типу T у програмі можна замінити об’єктами типу S без зміни будь-яких бажаних властивостей цієї програми.



Коли **дочірній** клас не може виконувати ті самі дії, що й його **батьківський** клас, це може викликати помилки.

Якщо у вас є клас і ви створюєте з нього інший клас, він стає батьківським **,** а новий клас стає дочірнім **.**Дочірній клас повинен мати можливість робити все, що **може** робити **батьківський клас.**Цей процес називається **успадкуванням** .

Дочірній клас повинен мати можливість обробляти ті самі запити та надавати той самий результат, що й **батьківський клас** , або він може надавати результат того самого типу.

На малюнку показано, що **батьківський** клас доставляє каву (це може бути будь-який тип кави). **Для дочірнього** класу прийнятно подавати капучіно, оскільки це особливий тип кави, але НЕ можна подавати воду.

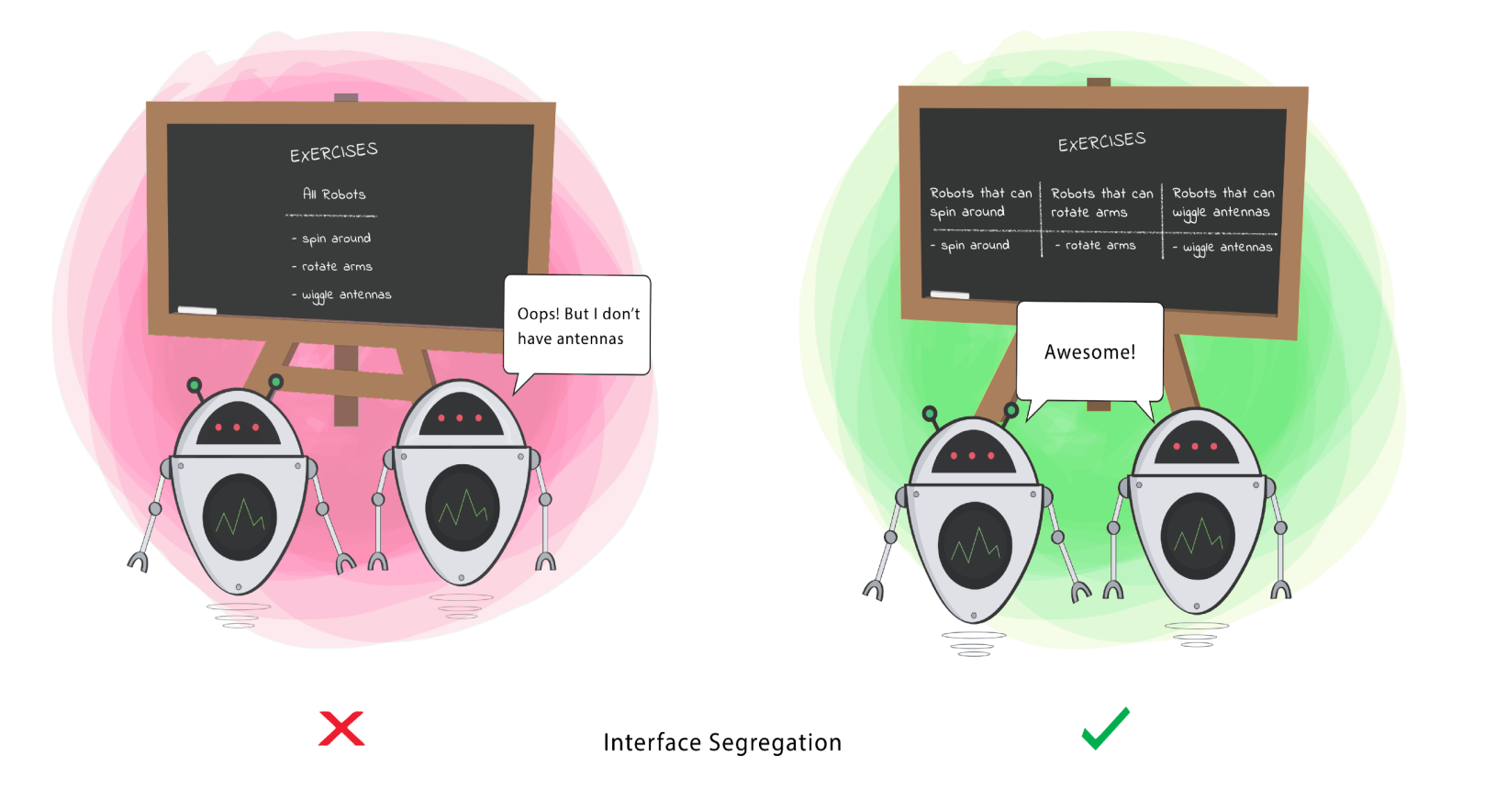
Якщо **дочірній** клас не відповідає цим вимогам, це означає, що **дочірній** клас повністю змінено та порушує цей принцип.

**Мета**

Цей принцип спрямований на забезпечення узгодженості, щоб батьківський клас або його дочірній клас могли використовуватися однаково без будь-яких помилок.

**I** — **Сегрегація інтерфейсу**

Клієнти не повинні бути змушені залежати від методів, які вони не використовують.



Коли від класу вимагається виконувати дії, які не є корисними, це марнотратно і може спричинити несподівані помилки, якщо клас не має можливості виконувати ці дії.

Клас повинен виконувати лише ті дії, які необхідні для виконання його ролі. Будь-яку іншу дію слід повністю видалити або перемістити в інше місце, якщо вона може використовуватися іншим класом у майбутньому.

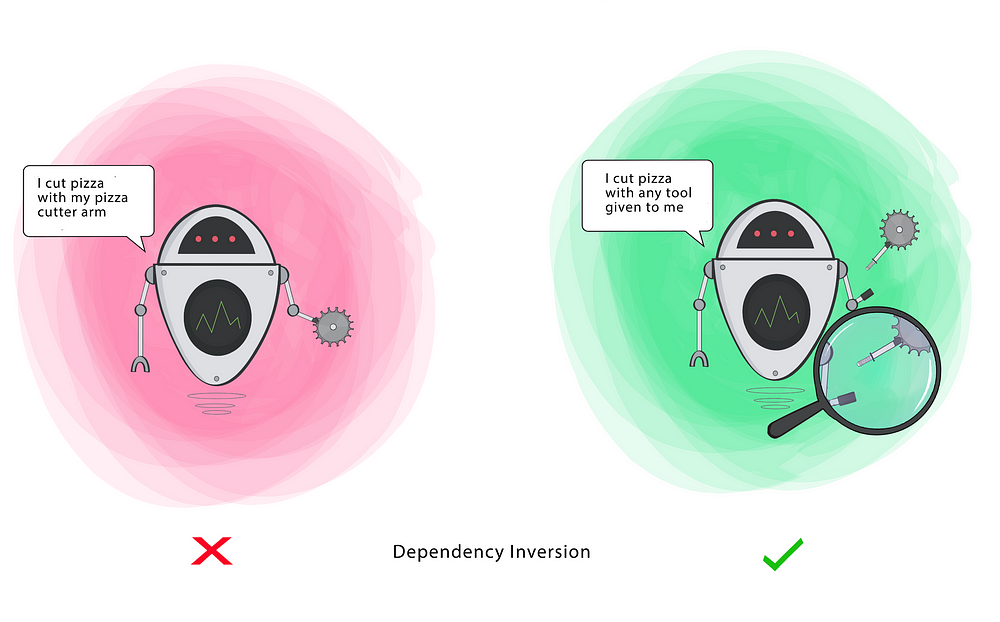
**Мета**

Цей принцип спрямований на розбиття набору дій на менші набори, щоб клас виконував ЛИШЕ той набір дій, який йому потрібен.

**D** — **Інверсія залежностей**

- Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва мають залежати від абстракції.

- Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.



По-перше, давайте простіше визначимо використані тут терміни

**Модуль (або клас) високого рівня** : клас, який виконує дію за допомогою інструменту.

**Модуль (або клас) низького рівня** : інструмент, необхідний для виконання дії

**Абстракція** : представляє інтерфейс, який з’єднує два класи.

**Подробиці** : Як працює інструмент

Цей принцип говорить, що клас не повинен бути злитий з інструментом, який він використовує для виконання дії. Натомість його слід об’єднати з інтерфейсом, який дозволить інструменту підключитися до класу.

Там також сказано, що як клас, так і інтерфейс не повинні знати, як працює інструмент. Однак інструмент має відповідати специфікаціям інтерфейсу.

**Мета**

Цей принцип спрямований на зменшення залежності класу високого рівня від класу низького рівня шляхом впровадження інтерфейсу.

**Резюме**

Наразі ми обговорили ці п’ять принципів і висвітлили їхні цілі. Вони мають допомогти вам зробити ваш код легким для налаштування, розширення та тестування без проблем.