ФІЛОСОФІЯ ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Питання 6.2.

Принцип DRY (DON'T REPEAT YOURSELF – не повторюйся)

- Базовий принцип розробки програмного забезпечення, націлений на зменшення повторень інформації.
  - Також відомий як DIE – Duplication Is Evil – дублювання – це зло.
  - Принцип DRY стверджує, що «Кожна частина даних (knowledge) повинна мати єдине, однозначне представлення в системі».
  - Порушення принципу DRY: "we enjoy typing" або "waste everyone's time “.
- Щоб досягти відповідності принципу, розділіть програмну систему на частини.
  - Не пишіть довгих методів, розбивайте по змозі логіку на менші частини, які можна повторно використати, та намагайтесь використовувати існуючі частини в своїх методах.
  - Вдалі приклади застосування принципу DRY: Enterprise-бібліотеки, helper-класи.
  - Кожна частина коду унікальна в них.

KISS - KEEP IT SIMPLE STUPID

- Тримайте код простим та прямолінійним для розуміння людиною.
- Тримайте методи маленькими, кожний метод має бути не більше 40-50 рядків коду.
  - Кожний метод має вирішувати тільки одну невелику задачу, а не багато use cases.
  - Якщо в методі багато умов, розбийте їх реалізацію в менші методи. Це спростить читання коду та пошук багів.
  - Яка з реалізацій методу зрозуміліша?

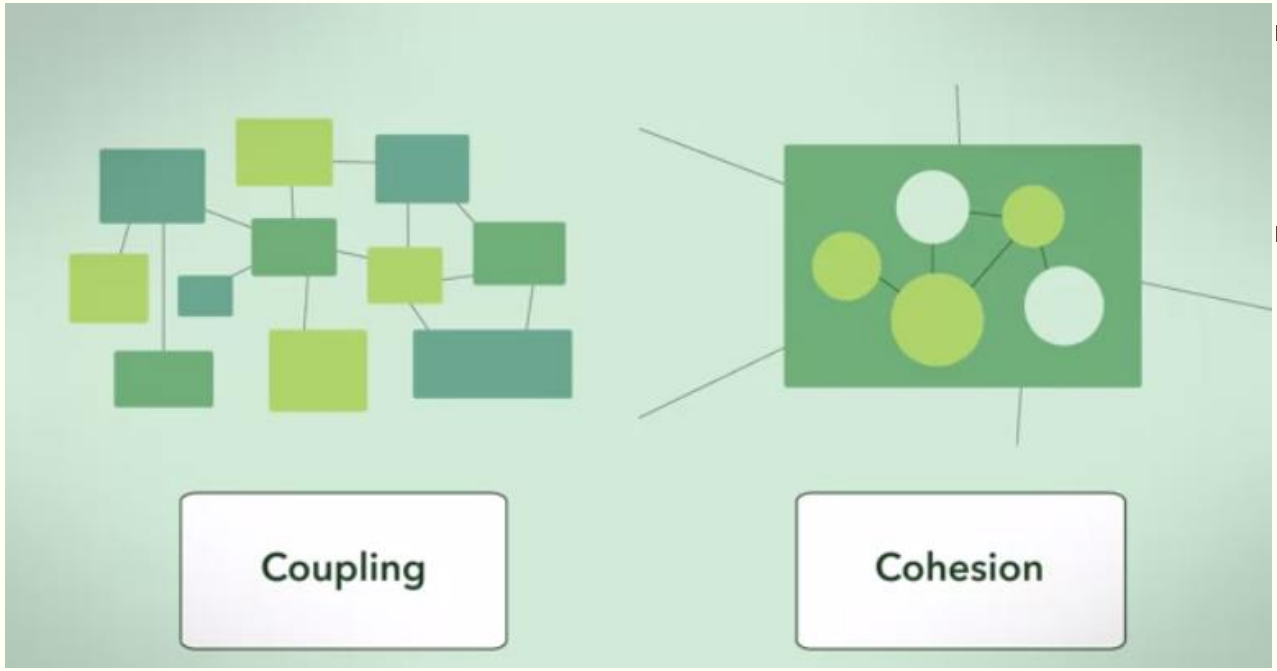
```
public String weekday1(int day) {
    switch (day) {
        case 1:
            return "Monday"; break;
        case 2:
            return "Tuesday";  break;
        case 3:
            return "Wednesday";  break;
        case 4:
            return "Thursday";  break;
        case 5:
            return "Friday";  break;
        case 6:
            return "Saturday";  break;
        case 7:
            return "Sunday";  break;
        default:
            throw new InvalidOperationException("day must be in range 1 to 7");
    }
}
```

```
public String weekday2(int day) {
    if ((day < 1) || (day > 7))
        throw new InvaidOperationException("day must be in range 1 to 7");
    string[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };
    return days[day - 1];
}
```

YAGNI - You aren't gonna need it

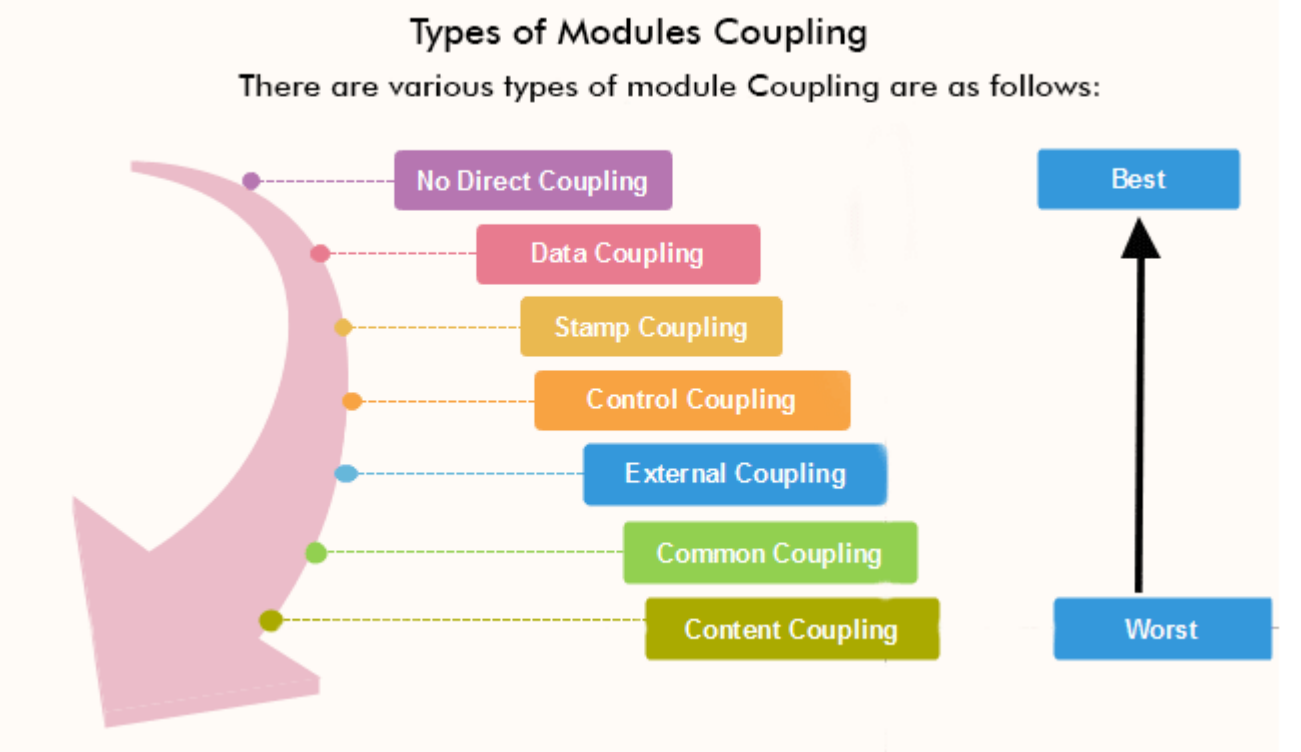
- Принцип екстремального програмування – XP.
  - XP використовується в процесі гнучкої (Agile) розробки програмного забезпечення.
- YAGNI говорить: не додавайте нової функціональності, поки в ній не буде нагальної потреби.
  - Іншими словами, пишіть код, який потрібний зараз, у поточній ситуації.
  - Не додавайте нічого, що ви думаєте може вам знадобитись.
  - Додавайте логіку в коді на даний момент, не думайте про можливі потреби в майбутньому.

Low Coupling та High Cohesion



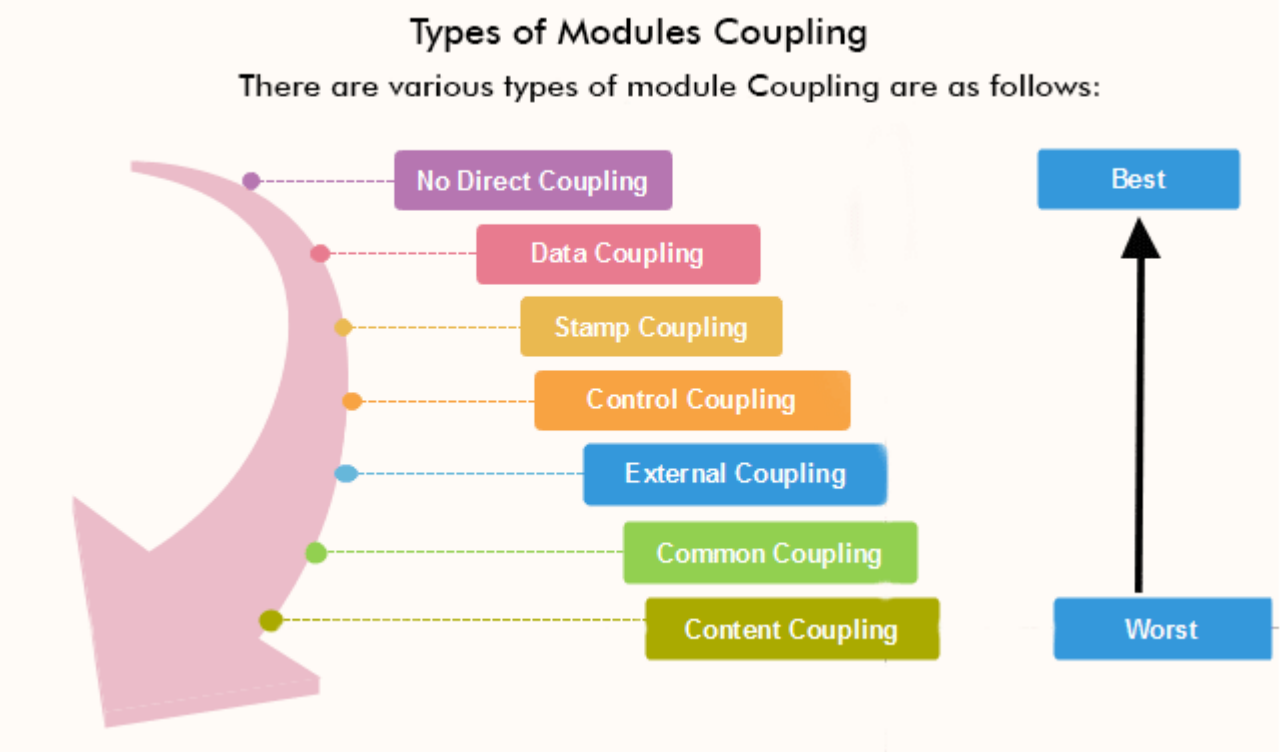
- Якісний дизайн володіє ***слабкою зв'язаністю (low coupling)*** і ***сильною зв'язністю (high cohesion)***.
- Зв'язаність, спряження (coupling)— спосіб і ступінь взаємозалежності між програмними модулями; сила взаємозв'язків між модулями; міра того, наскільки взаємозалежні різні підпрограми або модулі.
  - Сильна зв'язаність (High coupling) розглядається як серйозний недолік, оскільки ускладнює розуміння логіки модулів, їх модифікацію, автономне тестування, а також повторне використання окремо один від одного.
  - Слабка зв'язаність (Low coupling), навпаки, є ознакою добре структурованої та спроєктованої системи. Коли вона комбінується з сильною зв'язністю (high cohesion), це відповідає загальним показникам хорошої читабельності та супроводжуваності коду.

Low Coupling та High Cohesion



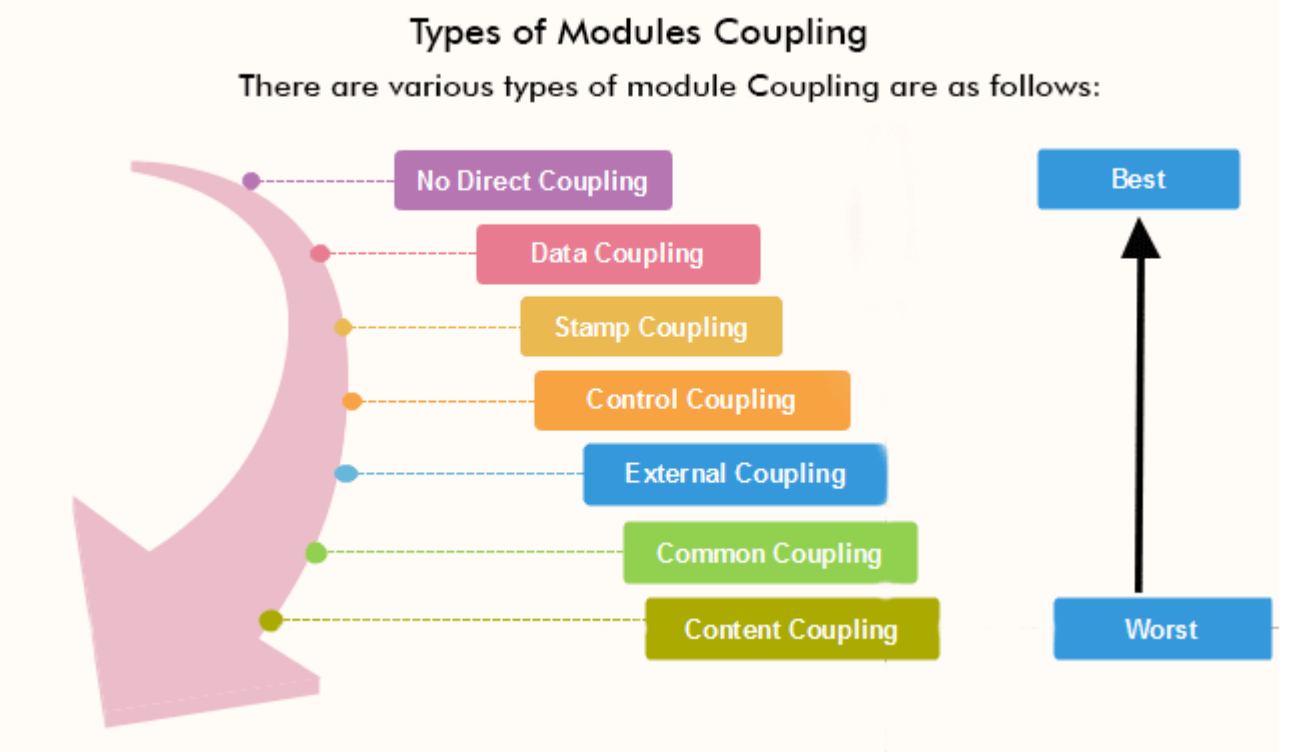
- **Low Coupling** — це принцип, який дозволяє розподілити обов'язки між об'єктами таким чином, щоб степінь зв'язаності між системами залишалась низькою.
  - **Степінь зв'язаності (coupling)** — це міра, що визначає, наскільки жорстко один елемент пов'язаний з іншими елементами, або якою кількістю даних про інші елементи він володіє.
  - Елемент з низькою степінню зв'язаності (слабким зв'язуванням) залежить від не дуже великої кількості інших елементів і має наступні властивості:
    - Мала кількість залежностей між класами (підсистемами).
    - Слабка залежність одного класу (підсистеми) від змін в іншому класі (підсистемі).
    - Висока степінь повторного використання підсистем.

Типи зв’язаності модулів



- **Зв’язаність вмісту (content coupling)**
  - Один модуль змінює або покладається на внутрішні особливості іншого модуля (наприклад, використовує локальні дані іншого модуля)
  - Зміна роботи другого модуля призведе до переписування першого
- **Зв’язаність через спільне (common coupling)**
  - Два модуля працюють і спільними даними (наприклад, глобальною змінною)
  - Зміна спільного ресурсу призведе до зміни всіх працюючих з ним модулів
- **Зв’язаність через зовнішнє (external coupling)**
  - Два модуля використовують нав’язаний ззовні формат даних, протокол зв’язку і т.д.
  - Зазвичай виникає через зовнішні сутності (інструменти, пристрої і т.д.)
- **Зв’язаність по керуванню (control coupling)**
  - Один модуль управляє поведінкою іншого
  - Присутня передача інформації про те, що і як робити

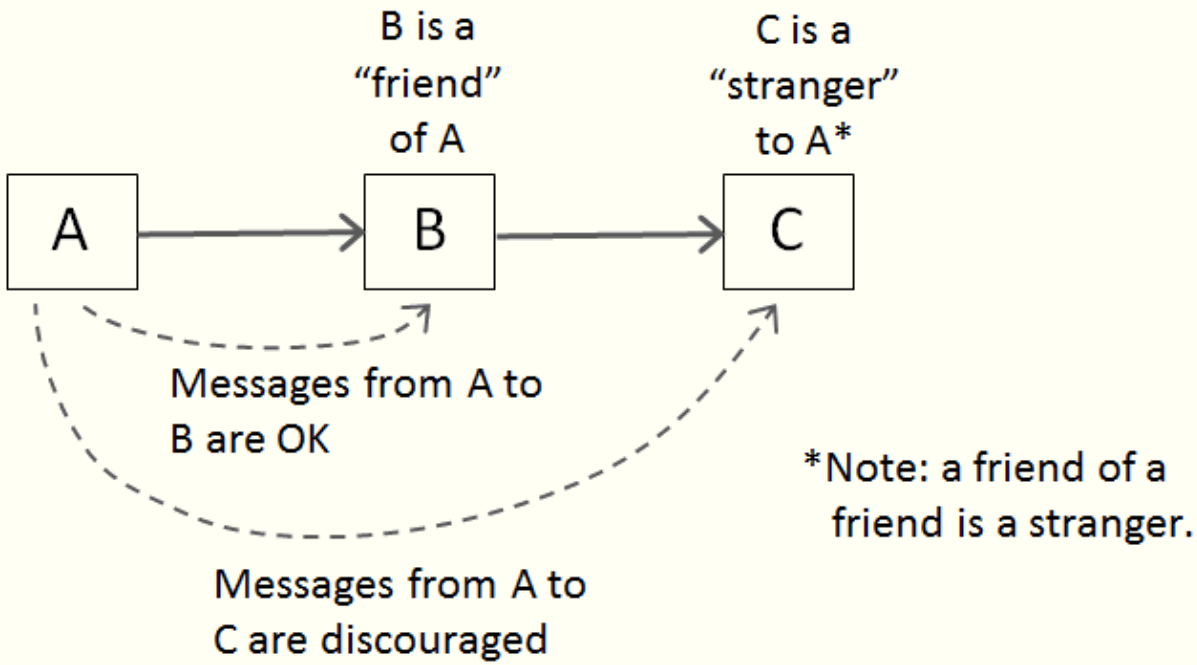
Типи зв’язаності модулів



- **Зв’язаність по структурованих даних (data-structured coupling, stamp couplig)**
  - Модулі використовують одну і ту ж структуру, проте кожний використовує тільки її частини.
  - Зміна структури може призвести до зміни модуля, який змінену частину навіть не використовує.
- **Зв’язаність через дані (data coupling)**
  - Модулі спільно використовують дані, наприклад, через параметри
  - Елементарні фрагменти маленькі, і тільки вони використовуються модулями спільно.
- **Зв’язаність по повідомленнях (message coupling)**
  - Модулі спілкуються тільки через передачу параметрів або повідомлень.
  - Стан децентралізований.
- **Відсутність зв’язаності (no coupling)**
  - Модулі взагалі ніяк не взаємодіють

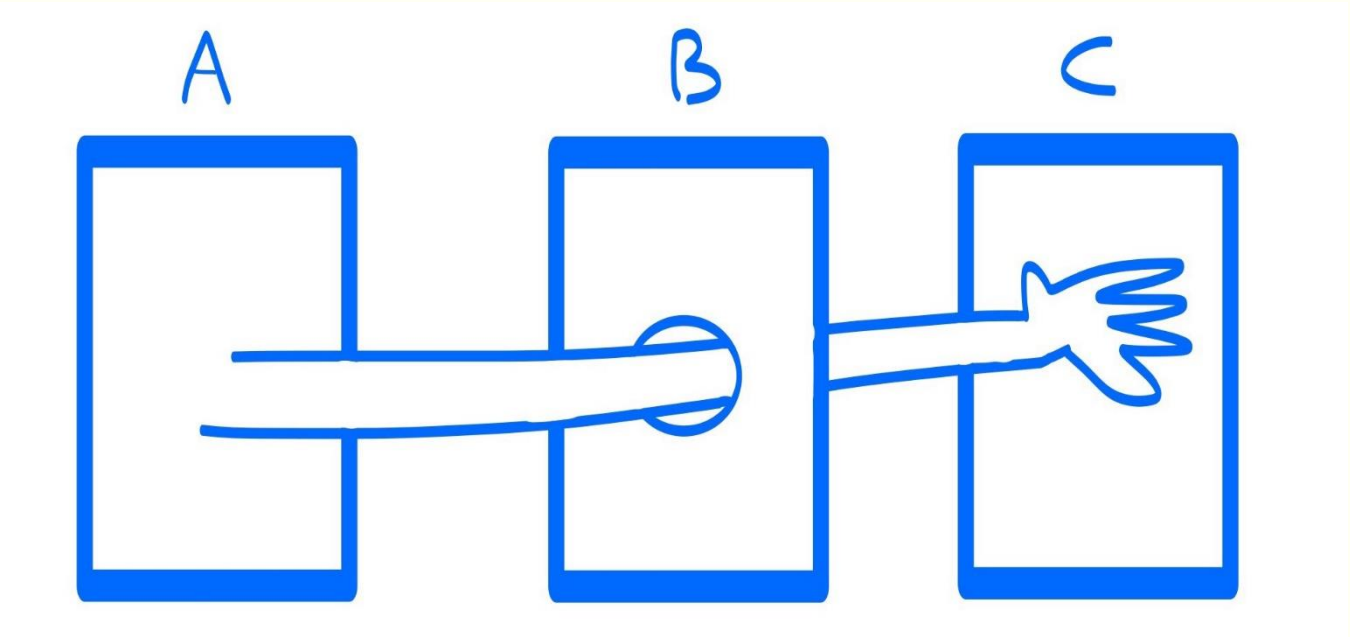
Закон Деметри

- Принцип найменшого знання



- Закон Деметри (Law of Demeter, LoD) — набір правил проєктування при розробці ПЗ, зокрема об’єктно-орієнтованих програм, який накладає обмеження на взаємодії об’єктів (модулів).
  - Узагальнено, закон Деметри є частинним випадком слабкої зв’язаності (loose coupling).
  - Правила були запропоновані наприкінці 1987р. в Північно-східному Університеті (Бостон, Массачусетс, США).

Закон Деметри

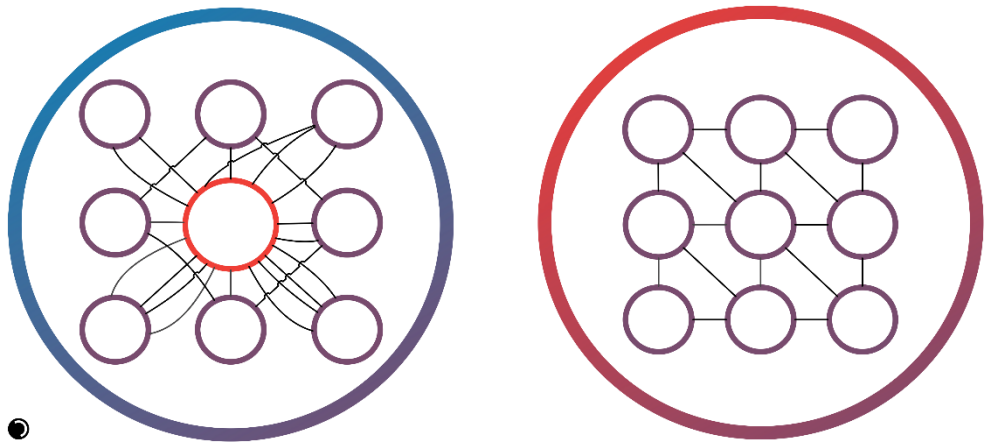


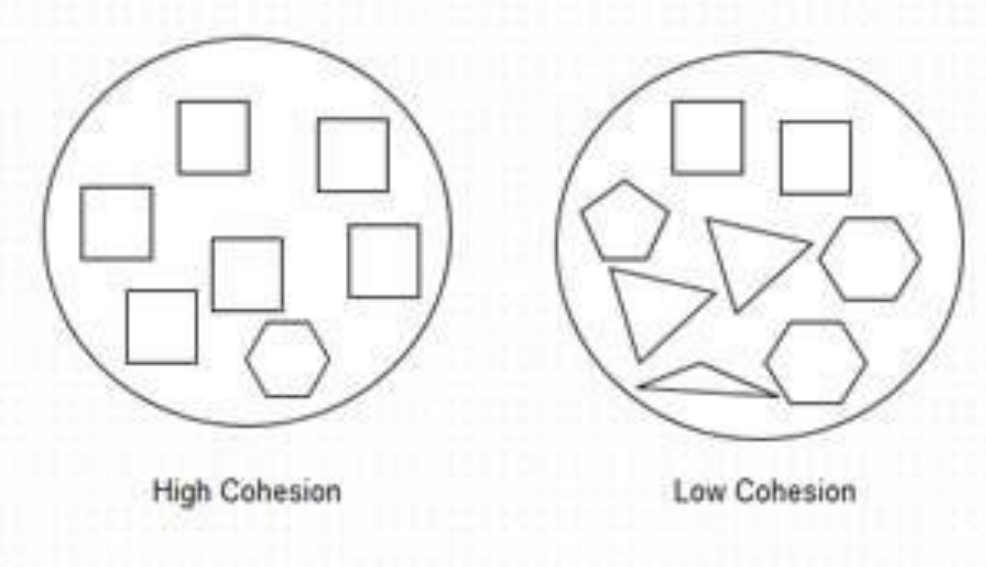
- Спрощено, кожний програмний модуль повинен:
  - володіти обмеженим знанням про інші модулі: знати про модулі, які мають «безпосереднє» відношення до цього модуля.
  - взаємодіяти тільки з відомими йому модулями-«друзями», не взаємодіяти з незнайомцями.
  - звертатись тільки до безпосередніх «друзів».
- Загальний опис правила: ***Об'єкт A не повинен мати можливість отримати безпосередній доступ до об'єкта C, якщо в об'єкта A є доступ до об'єкта B, а в об'єкта B є доступ до об'єкта C.***
  - Таким чином, код a.b.Method() порушує закон Деметры, а код a.Method() є коректним.

Закон Деметри

- Переваги:
  - Код, розроблений з дотриманням закону, спрощує написання тестів.
  - Розроблене ПЗ менш складне при супроводі та має більші можливості для повторного використання коду.
  - Оскільки об’єкти є менш залежними від внутрішньої структури інших об’єктів, контейнери об’єктів можуть буть змінені без модифікації викликаючих об’єктів (клієнтів).
- Недолік:
  - інколи потрібно створювати велику кількість малих методів-адаптерів (делегатів) для передачі викликів методу до внутрішніх компонентів.

Зв'язність (Cohesion)





- **Зв'язність**, або **міцність** (**cohesion**) — міра сили взаємопов’язаності елементів всередині модуля; спосіб і степінь, у якій задачі, що виконуються деяким програмним продуктом, пов’язані одна з одною.
- *Зв’язність характеризує те, наскільки добре всі методи класу або всі фрагменти методу відповідають головній меті, — іншими словами, наскільки сфокусований клас.*

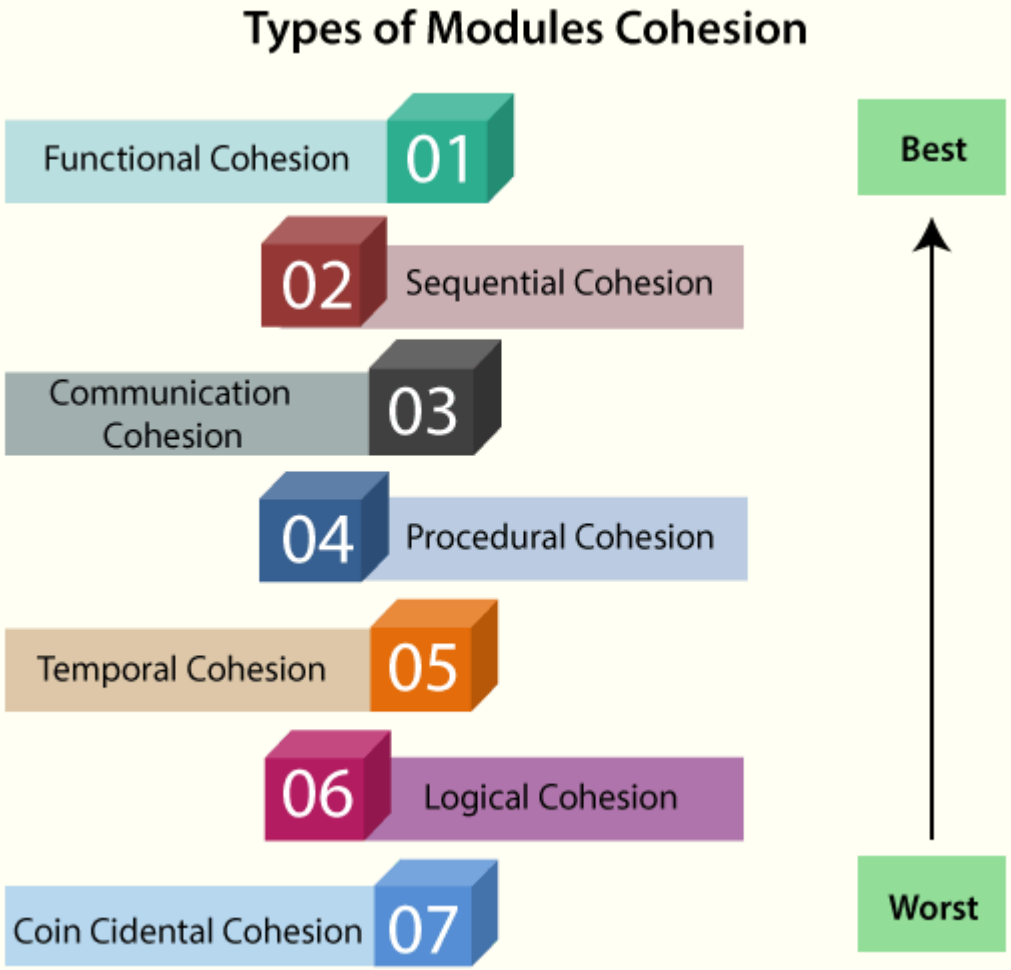
Стив Макконнелл

- Вважається, що об’єкт (підсистема) володіє високою зв'язністю (High cohesion), якщо його обов’язки добре узгоджені між собою, і він не виконує величезних об’ємів роботи.

Зв'язність (Cohesion)

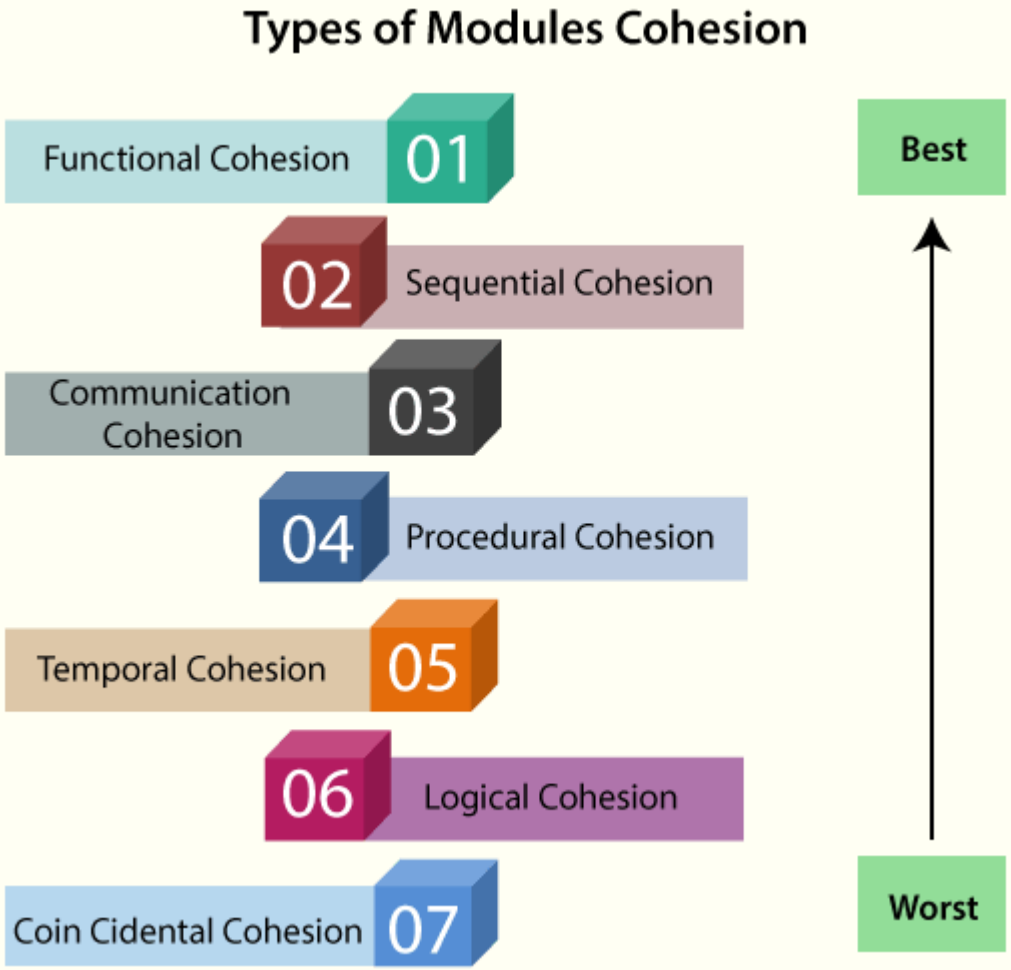
- Клас з **низькою зв'язністю (low cohesion)** виконує багато різнорідних функцій або непов’язаних між собою обов’язків.
- Такі класи небажано створювати, оскільки вони призводять до виникнення наступних проблем:
  - Складність розуміння
  - Складність повторного використання
  - Складність підтримки
  - Ненадійність, постійна схильныcть до змін
- Класи з низькою степінню зв’язності, як правило, є надто «**абстрактними**» або виконують обов’язки, що легко можна розподілити між іншими об’єктами.

Види зв'язності



- **Випадкова (coincidental cohesion)**
  - Частини модуля згруповані "від ліхтаря"
  - Едине, що їх об'єднує, - сам модуль.
- **Логічна (logical cohesion)**
  - Частини модуля логічно відносяться до однієї проблеми
  - При цьому частини можуть відрізнятись за своєю природою.
- **Часова (temporal cohesion)**
  - Частини модуля зазвичай використовуються в програмі одночасно, поряд.
- **Процедурна (procedural cohesion)**
  - Частини модуля завджи використовуються в певному порядку

Види зв'язності



- **За взаємодією (communication cohesion)**
  - Частини модуля працюють над одними й тими ж даними.
- **За послідовністю дій (sequential cohesion)**
  - Результат роботи однієї частини модуля є вхідними даними для іншої частини.
- **Функціональна (functional cohesion)**
  - Частини модуля направлені на вирішення однієї чіткої задачі, за яку відповідає модуль

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: SOLID-принципи розробки об’єктно-орієнтованого коду