Принципи за гъвкъв и подръжаем обектно-ориентиран дизайн
(Java базирани примери)

S.O.L.I.D дизайн принципи:

1. **S** – **Single Responsibility** Principle
2. **O** – **Open-Closed** Principle
3. **L** – **Liskov Substitution** Principle
4. **I** – **Interface Segregation** Principle
5. **D** – **Dependency Inversion** Principle

1. **Single Responsibility Principle.** Всеки клас трябва да отговаря само за едно нещо. Казано по друг начин – ако се променят изискванията по нашия софтуер, един клас трябва да има една единствена причина да се променя.

   Пример за нарушаване на Single Responsibility принципа:
   Нарушаване на наслояването на нашето приложение:
   - изпълнение на бизнес логика в слоя за достъп до данни, чиято единствена отговорност трябва да бъде осигуряването на постоянен достъп до приложението;
   - достъп на бизнес услуги от домейн слоя, който единствено трябва да отговаря за съхраняването на по-голямата част от състоянието на приложението;
   - изпълнение на по-сложна бизнес логика в слоя за изглед, който е отговорен за представянето на данни и въвеждане на такива от потребителя.

2. **Open-Closed Principle.** Софтуерните единици (класове, модули, функции и т.н.), трябва да са отворени за разширяване и затворени за модификация. Това означава, че след време трябва да може да добавяме функционалност в софтуера, без да разбутваме наличните вече функционалности.

   Пример за нарушаване на Open-Closed принципа:
   Представете си, че стартирате свой бизнес за облекла и се изисква да създадете система за инвентаризация, за да управлявате инвентара си от дрехи. Първоначално сте заредили само с тениски.
   Създавате клас Inventory и клас Tshirt. В Inventory имате член данна за колекция от Tshirt и метод за добавяне на тениска в нея, а в Tshirt имате член данна за цена и метод за връщане на тази цена.
   След време, бизнеса ви просперира и зареждате и шапки. Следователно правим клас шапка, който има член данна за цена и метод, който да връща тази цена, също като тениската. В Inventory, обаче, се налага да правим нова колекция, която да съхранява шапките, тъй като старата съхранява тениските, както и нов метод за добавяне в тази колекция. Това е пример за нарушаване на Open/Closed принципа, тъй като разширихме нашето приложение, но се наложи да модифицираме съществуващ клас.

   За да избегнем това, трябва да използваме интерфейс Wearable, например, като слой за абстракция, който да се имплементира и от Tshirt и от Hat класовете и в Inventory да имаме колекция от Wearable. Така няма да ни се налага да правим отделни методи за добавяне на различни облекла в колекцията. Ще разчитаме на принципите на абстракцията и полиморфизма.

3. **Liskov Substitution Principle.** Обектите в една програма на обектно ориентиран език, трябва да са заменими с инстанции на техни класове наследници без това да променя коректността на програмата.

   Индикатори за това, че не сме спазили Liskov Substitution принципа:
   - Хвърлянето на NotImplementedException от клас наследник.
   - Скриването на виртуален базов метод от метод на дериватен клас, който използва ключовата дума new. Трябва да използваме @Override.
   - Връщането на тип данни с рестрикции, които не се знаят от сигнатурата на метода. Например връщането на ReadOnlyCollection от дериватен клас, докато в базовият

клас връщаме Collection.

4. **Interface Segregation Principle.** Клиент (парче код, което използва нашия код) не трябва да бъде принуждаван да имплементира метод, които не използва. По-добре да имаме множество специфични интерфейси, отколкото един интерфейс тип швейцарско ножче.

   Пример за нарушаване на Interface Segregation принципа:
   Представете си, че сте направили софтуер за паркинг система, който ще вдига и сваля някаква бариера (т.е. ще има механична част), ще води статистика за свободните и заетите места (т.е. ще има някакво състояние) и ще пресмята сумата за престоя на даден автомобил.

   Паркинг системата се харесва от редица молове и те си я закупуват. Системата работи и всички са доволни. Но в бъдеще идва клиент, който харесва системата, но неговият паркинг е безплатен. Не е редно да преправяме кода и да даваме dummy имплементация за неизползваните методи, а просто да премахнем интерфейсите, който не са обходими на клиента. Т.е. само да ограничим поведението на системата.

5. **Dependency Inversion Principle.** В едно приложение, модулите от по-високо ниво, не трябва да зависят от модулите от по-ниско ниво, а да зависят от някакви абстракции. Т.е. трябва да зависят от абстракции/интерфейси, а не от конкретна имплементация от ниско ниво.

   Пример за нарушаване на Dependency Inversion принципа:
   Представете си, че имаме някакъв клас, който се грижи за записването на някаква информация. В класът има метод write(...), който осъществява логиката за записването, но той приема път до файл във файловата система. Сега, ако искаме да запаметим информацията в някакъв друг тип носител, като например база от данни, ще се наложи да преправяме метода.

   За да може да избегнем този проблем трябва да направим още по-абстрактен метода и да му подаваме обект, в който може да се пише. Т.е. обекта да поддържа поведението да може да се пише в него. Тогава, ако искаме да се сменя къде да се съхранява индекса/информацията, само ще подаваме друга инстанция като аргумент, без да преправяме кода в метода.

   Още един пример за нарушаване на принципа:
   Например, ако искаме да връщаме някаква колекция, от която се интересуваме само от итерация по елементите ѝ, да връщаме в дадения метод List вместо Collection.