SOLID и други принципи

SOLID:

SRP = Single responsibility principle - гласи че един клас, един метод, та дори една променлива, трябва да има една единствена отговорност. SRP is about strong cohesion and loose coupling. Класовете трябва да правят едно нещо и да depend-ват колкото се може по-малко на други класове

Classic violations of SRP

* Objects that can print/ draw themselves – например ако героите от дадена игра са се рисували в дадена 2D среда, а след това искаме да рисуваме тези герои в 3D среда, то трябва да минем през имлементацията на всички класове, които представляват тези герои и да променим имплементацията на метода Draw() или метода Print() например. За това вместо героите да могат да рисуват сами себе си, т.е. да имат отделен метод Draw() или Print(), то трябва всеки герой да приема някакъв обект, имплементиращ интерфейс IDraw или IPrint, който рисува герои
* Objects that can save/ restore themselves – ако даден клас знае как да си пази данните и да се restore-ва
* Един потребител, например, не трябва да може да се записва сам в базата данни. Това е отговорност на клас като UsersRepository например

OCP = Open/ Closed principle – едно парче код, програма, приложение трябва да бъде отворено към разширяване, но затворено към промени. Става като наследяваме един клас и override-ваме virtual-ните му членове, които искаме да променяме, и добавяме свои допълнителни функционалности в класа наследник;

Начини за постигане на OCP – 1)чрез делегати, 2)наследяване на базовия клас и имплементиране на нашите си функционалности в класовете наследници, 3)Composition/ Strategy patterns – трябва да работим през интерфейси, а не през конкретни класове. Така може да подаваме различни класове на структурата, която работи с интерфейси

Заради Open/ Closed принципа дизайнът се усложнява, но поддръжката в бъдеще става по-лесна, особено при големи проекти

LSP = Liskov substitution principle – ако имаме един базов клас, например Person, и имаме даден тип данни, който работи с Person-и то този тип данни трябва да може да работи и с наследници на класа Person, без да се получават грешки при работата с различните наследници на Person класа. Един от начините за да не допускаме грешки е да разделяме различните функционалности на различни по-малки интерфейси/класове, а не всички функционалности да бутаме в един голям интерфейс/клас

ISP = Interface segregation principle – ако имаме един дебел интерфейс, който дефинира много неща, то трябва да раздробим този интерфейс на няколко по-малки интерфейси

DIR = Dependency inversion principle - един компонент от нашия код не трябва да depend-ва на нещо конкретно, а на някаква абстракция (абстрактен метод или интерфейс). Постига се чрез Dependency injection

DRY = Don’t repeat yourself – ако дадено парче код се повтаря много пъти, то трябва да изнесем това парче код в отделен метод

YAGNI = не трябва да пишем неща, които на момента не са нужни

KISS = кодът трябва да е максимално опростен