# Types of Design Patterns

* Представляват решения на често срещани проблеми в обектно ориентирания дизайн.
* **Елементи на дизайн патерна**

1. Име, което е общо прието.
2. Причина да бъде използван.
3. Решение - с абстрактен код.

* Последствия - носи и някакъв минус. Примерно време за направата му. Води до повече редове код в някои ситуации. Усложнява се кода.
* **Ползи**

1. можем по бързо да говорим, на по-високо ниво на абстракция. Казваме повече неща с по-малко думи.
2. можем да преизползваме вече доказали се решения в практиката.
3. подобрява се качеството на кода, което позволява след това по-лесно да се вкара нова функционалност.

* **Минуси**

1. не водят до директна преизполваемост на кода.

* **Creational** - занимават се основно със създаването на обекти. Как да си подредим кода, какви класове трябва да направим.
* **Singleton pattern** - основната му цел е да забрани инстанцирането на един клас на други места и да ни даде възможност да използваме една единствена негова инстанция.
* **Simple Factory** - получава параметър или параметри и на базата на тези параметри, връща инстанция на нещо.
* **Factory Method** – позволява ни да вдигнем малко абстракцията върху simple factori-то. Използва се много често в настройките. Определя под класовете какви точно ще бъдат.
* **Abstract Factory** – вдига още нивото на абстракция. Можем да създаваме много различни неща, които дори да не са от една и съща фамилия от класове. Примерно единия метод може да връща, телефони, другия животни.
* **Builder** – използваме го, когато имаме стъпки за да построим нещо. Когато искаме до уточним списък от стъпки и точната им последователност.
* **Prototype** – Използва се за клониране на обекти. Получаваме точно копие на вече съществуващ обект.
* **Fluent Interface** – след последователност от извикване на някакви методи, да имаме продуциран обект. Пишем нещо, натискаме точка и продължаване с друго нещо. Примерно Html.Grid().Columns() …. Това всъщност е **LINQ**.
* **Object Pool** – използва се когато създаването на нов обект струва „скъпо“. Веднъж създаден обекта, ако за момента не е нужен, той се съхранява докато не потрябва пак. „Скъпо“ е например създаването на връзка към базата данни.
* **Lazy Initialization** – първия път, когато ни потрябва даден обект, той се създава и си остава създаден до края. Създава се в момента в който го поискаме, а не в началото.
* **Structoral** - занимават се с това как комбинираме и структурираме вече съществуващи класове, така, че да достигнем до някаква нова функционалност. Свързано е с композицията на класове и обекти.
* **Façade** – изкарва допълнителен клас или класове, които комбинират вече съществуващите класове и изкарва от тях основните методи. Пример – устройството за гледане на телевизия. Скрива сложна функционалност зад елементарни за ползване класове.
* **Composite** – позволява създаването на дървовидни структури.
* **Flyweight** – позволява преизползването на вече готови неща. Например когато имаме букви, не се налага всеки път да описваме как се рисува буквата, когато я пишем. В **.NET** когато направим **var a = “text”** и **var b = “text”**, понеже стринга и в двете променливи е еднакъв, той се преизползва. Не се пази двойна памет. И двете променливи сочат към едно и също място в паметта.
* **Proxy** – Извършва допълнителни (междинни) действия преди да прехвърли управлението на оригинални клас. Примерно проверява дали имаме необходимата наличност в сметката преди да ни даде да теглим. Пази оригиналния клас в себе си и преди да извика някакви методи на оригиналния клас, извършва някаква друга работа.
* **Decorator** – Разширява съществуваща функционалност да даден клас. Могат да се добавят много декоратори един върху друг. Типичен пример са стримовете в .NET. Имаме **Stream()** който чете и пише, но имаме и **GzipStream()**, който приема във себе си Stream - **GzipStream(**Stream**)**.
* **Adapter** – Клас, който имплементира наш интерфейс, като методите от нашия интерфейс ги пренасочва към някакъв друг интерфейс или обект до който нямаме достъп.
* **Bridge** – Вдига абстракцията. Изнася част от кода в друга йерархия от класове. Може по-лесно да се подменя съществуваща функционалност с него и подобрява качеството на кода.
* **Behavioral** - целят да постигнат ново поведение или комбинация от поведения. Концентрират се върху това какво върши нещо.
* **Chain of Responsibility** – Прилича на Composite патерна, но може да вземе решение дали да обработи заявката, дали да я даде на някой от наследниците или на никой наследник. Представлява дървовидна структура върху обекти. Примерно при йерархична структура на работници и служители, започваме от този с най-ниската йерархия – даваме му работа и той преценява дали може да свърши тази работа. Ако може, я върши. Ако неможе я подава на този, който е над него. При този патерн знаем само за един обект, който се намира след/пред нас, не за няколко, както е при Composite патерна.
* **Iterator** – Скрива в себе си информацията как се минава от елемент в елемент. Foreach използва итератор за да минава от елемент в елемент.
* **Command** –
* **Template Method** –
* **Strategy** – Имплементира интерфейс с един метод. Този интерфейс задава някакво действие. Изнася базов интерфейс и прави негови наследници. Пример – алгоритмите за сортиране го ползват.
* **Observer** – Дава решение на това как да се закачаме към събития. Когато възникне събитие, всички закачени към това събитие ивент хендлъри ще бъдат извикани един след друг в реда в който са закачени.