7. Létrehozási tervezési minták

# Factory (method) (Creational pattern)

* A Factory Method lehetővé teszi, hogy az új példány létrehozását leszármazott osztályra bízzuk. (Szokás virtuális konstruktornak is nevezni.)
* **Probléma**
  + Az objektumainkat gyakran bonyolult létrehozni és a konstruktor nem elég flexibilis ehhez.
* **Megoldás**
  + Az új objektumainkat a factory method-on belül hozzuk létre, ha pedig vissza is tér ezzel az objektummal, akkor azokat product-oknak is szokták nevezni.

## Factory használjuk, ha

* Egy osztály nem látja előre annak az objektumnak az osztályát, amit létre kell hoznia.
* Egy osztály azt szeretné, hogy leszármazottjai határozzák meg azt az objektumot, amit létre kell hoznia.

## Factory implementálása

1. Interfész implementálása a megfelelő metódusok segítségével.
2. A creator osztályban adjunk hozzá egy üres factory method-ot, ami visszatér az interfész típusával.
3. Factory method-ban hozzuk létre az új objektumokat.
4. Creator alosztályokat hozunk létre, ami a megfelelő factory method-ot használja.
5. Ezek után a base factory method üressé válik, így ezt abstract-á tehetjük.

## Factory előnyök és hátrányok

* **Előnyök**
  + Single Responsibility elv
  + Open/Closed elv
* **Hátrányok**
  + A sok alosztály miatt bonyolulttá válhat a kód.

# Abstract Factory (Creational pattern)

* **Probléma**
  + Különböző feltételek alapján más és más objektumokat szeretnénk szolgáltatni.
    - Pl egy stringtől függ, hogy milyen osztályt példányosítunk.
* **Megoldás**
  + Egy ősfactory – sok leszármazott factory
  + Dictionary vagy reflexió azonosítja a paraméter függvényében a megfelelő factory-t.

## Abstract factory használjuk, ha

* A rendszernek függetlennek kell lennie az általa létrehozott dolgoktól.
* A rendszernek több termékcsaláddal kell együttműködnie.

## Abstract factory előnyök és hátrányok

* **Előnyök**
  + Elszigeteli a konkrét osztályokat
  + Elősegíti a termékek közötti konzisztenciát.
* **Hátrányok**
  + Nehéz új termék hozzáadása.
    - Ilyenkor az Abstract Factory egész hierarchiáját módosítani kell, mert az interfész rögzíti a létrehozható termékeket.

# Builder (Creational pattern)

* Lehetővé teszi az összetett objektumok lépésről-lépésre történő létrehozását.
* **Probléma**
  + Van egy összetett objektum, ami számos mezőt és egymásba ágyazott objektumot tartalmaz, ami inicializálást igényel.
  + Egy ilyen inicializálási kód általában sok paramétert tartalmazó konstruktorban van elrejtve vagy még rosszabb, ha a kliens kódban vannak szétszórva.
  + **Túl bonyolulttá teheti a programot, ha egy objektum minden lehetséges konfigurációjára létrehoz egy alosztályt.**
  + **Túl sok paramétere van a konstruktornak, ez így nagyon csúnya.** (Lehet a paraméterek egy része nem is kell.)
* **Megoldás**
  + Az objektum létrehozásának kódját ne a saját osztályába rakjuk bele, hanem helyezzük át egy builder objektumba.

## Builder használati esetek

* Telescoping konstruktoroktól mentesség (pl.: Egy konstruktor egy paraméternek, másik konstruktor másik paraméternek, stb.)
* Objektum felépítése lépésről-lépésre.

## Builder implementálása

1. Határozzuk meg a builder lépéseit. (Pl.: Hogyan építsünk fel egy objektumot)
2. Base builder interfész kialakítása.
3. Builder osztály létrehozása, ami implementálja a builder interfészt.
4. Director osztály lérehozása.
   1. Különböző metódusokat tartalmazhat az objektumok létrehozására.
5. Kliens kód használja a builder és a director objektumokat.
   1. Először a builder objektumot át kell adni a director-nak konstruktoron keresztül paraméterként.
   2. Innentől kezdve a director használja a builder-t.
6. Builder eredmény akkor születik a director-ból, ha minden elem ugyanazt az interfészt használja.
   1. Ellenkező esetben a kliensnek az eredményt a builder-től kell lekérnie.

## Builder előnyök és hátrányok

* **Előnyök**
  + Lépésről-lépésre való „építkezés”/building.
  + Single Responsibility elv-et követi.
  + Komplex kód elkülönítése a business logic-tól.
* **Hátrányok**
  + A kód komplexitása növekszik, mivel több új osztály létrehozását igényli.

# Singleton (Creational pattern)

* Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni és ehhez az egy példányhoz globális hozzáférést biztosít.
* **Probléma**
  + Van egy objektumunk és egy idő után feltűnik, hogy ugyanazt az objektumot használtuk.
  + Globális változók lehet tárolnak fontos dolgokat, de mégis felül lehet írni kívülről.
* **Megoldás**
  + Legyen az osztály felelőssége, hogy csak egy példányt lehessen belőle létrehozni.
  + Biztosítson hozzáférést ehhez az egy példányhoz.
  + Az Instance osztály-művelet (statikus) meghívásával lehet példányt létrehozni, illetve az egyetlen példányt elérni.
* **Az Instance**
  + Mindig ugyanazt az objektumot adja vissza.
  + C# esetén property-vel célszerű: Singleton.Instance
* A Singleton konstruktora protected láthatóságú.
  + Ez garantálja, hogy csak a statikus Instance metódushíváson keresztül lehessen példányt létrehozni.

## Singleton használati esetek

* Ha egy osztálynak csak egyetlen példánya kell, hogy legyen, ami minden kliens számára elérhető. (Pl.: egyetlen adatbázis-objektum, amit a program különböző részei megosztanak.)
* Szigorúbb ellenőrzésre van szüksége a globális változók felett.

## Singleton implementálása

1. Privát statikus mező létrehozása az osztályban a singleton példány tárolására.
2. Nyilvános statikus létrehozási metódus deklarálása a singleton példány kinyeréséhez.
3. A statikus metóduson belül inicializálás végrehajtása.
   1. Első híváskor az új objektum létrehozása és statikus mezőbe helyezése.
   2. A metódusnak minden további híváskor mindig ezt a példányt kell visszaadnia.
4. Az osztály konstruktora legyen privát.
   1. Az osztály statikus metódusa továbbra is képes lesz meghívni a konstruktort, de a többi objektum nem.

## Singleton előnyei és hátrányai

* **Előnyei**
  + Egyetlen példánya van az osztálynak, globális pontot biztosít ehhez a példányhoz.
  + A singleton objektum csak akkor inicializálódik, amikor először kérjük.
* **Hátrányai**
  + Speciális kezelést igényel többszálas környezetben, hogy több szál ne hozzon létre többször egy singleton objektumot.
  + Nehezíti a Unit tesztelést, mock objektum előállítása nehézkes. Konstruktor privát.

# Prototype (Creational pattern)

* A prototípus alapján új objektumpéldányok készítése.
* Minden objektum támogatja (Object osztály művelete)
  + Shallow copy
* Igazi, publikus, mély másolatot végző klónozáshoz implementálható az ICloneable interfész
  + Deep copy
* **Probléma**
  + Átakarunk másolni minden egyes objektumot, de lehetnek olyan mezők, amik privátok, nem láthatóak kívülről.
  + Másik probléma, hogy mivel a duplikátum létrehozásához ismerni kell az objektum osztályát, a kód függővé válik az osztálytól.
* **Megoldás**
  + Létrehozunk egy interfészt, ami az összes objektumnak elérhető.
  + Ezáltal lehet klónozni, ami egy Clone metódus.
  + A metódus létrehoz egy objektumot az aktuális osztályból és a régi objektum összes mezőértékét átviszi az új objektumba. (Így már a privát mezők is másolhatóak.)
  + **Azaz objektum, ami támogatja a klónozást, azt hívjuk prototype-nak.**

## Prototype használjuk, ha

* Egy rendszernek függetlennek kell lennie a létrehozandó objektumok típusától.
* Ha a példányosítandó osztályok futási időben határozhatók meg.
* Ha nem akarunk nagy párhuzamos osztályhierarchiákat.
* Amikor az objektumok felparaméterezése körülményes és könnyebb egy prototípust inicializálni, majd azt átmásolni.

## Prototype implementálása

1. Prototype interfész létrehozása, amiben van egy Clone metódus vagy interfész nélkül.
2. A prototype osztálynak lennie kell egy alternatív konstruktornak, ami elfogadja az adott osztály egy objektumát.
   1. A konstruktornak az átadott objektumból az osztályban definiált összes mező értékét át kell másolnia az újonnan létrehozott példányba.
   2. Ha egy alosztályt változtatunk, akkor meg kell hívnunk a szülő konstruktort, hogy az ősosztályt kezelje a privát mezők klónozását.
3. Clone metódus felülírása new operátorral, ezáltal új logikát adhatunk neki.

## Prototype előnyök és hátrányok

* **Előnyök**
  + Objektumok hozzáadása és elvétele futási időben
  + Új, változó struktúrájú objektumok létrehozása
  + Redukált származtatás, kevesebb alosztály
* **Hátrányok**
  + Minden egyes prototípusnak implementálnia kell a Clone() függvényt, ami bonyolult lehet.