OOP elvek: egységbezárás, adatrejtés, öröklés, polimorfizmus

UML: Unified Modeling Language
- Cél: szoftverfejlesztés és dokumentáció modellezése
- Általános modellező nyelv
- Jelenlegi verzió: 2.5.1
- Jelölésrendszer, közös nyelv
- Magas absztrakciós szint, tömör leírás
- Szabványos keret
- Kezdőérték is lehet UML-be, csak a Modelio nem támogatja azt

UML:	**Szakterületi nyelvek:**
- Egységes, szabványos - Minden szoftvermérnök ismeri - Jó eszköztámogatás - Korlátozott kód generálás - Megrendelő számára érthetetlen	- Egyedi - Csak a szakértők ismeri - Egyedi eszközök kellenek - Akár teljes kódgenerálás - Megrendelő is jól érti

UML Profile:
- alapgondolat: UML kiterjesztése
- közös alapok
- eszköztámogatás
- szakterületi szabályok is jól leírhatók
- OMG szabvány (Object Management Group)
- sztereotípiák, kényszerek, tag-ek
- nem mond ellent az eredeti specifikációnak
- könnyű megtanulni

Pl. UML Profile for XML, MARTE (valósidejű rendszerekhez), SysML (rendszermodellezés)

Metamodellezés:
- a nyelvek leírását is egy modellben adjuk meg (modellezzük a modellt)
- rugalmasan összerakhatóak “saját” nyelvek
- nyelv leíró nyelv kell hozzá

pl. Meta-Object Facility (MoF):
- OMG szabvány
- UML általánosítása
- Complete MoF/Essential MoF

Önleíró, 4 szintű metamodellezési hierarchia:

0. szint: objektumok
1. szint: UML modellek
2. szint: UML modellek modellje
3. szint: MoF önleíró alapstruktúra

Kényszerek: szöveges nyelv a grafikus modellhez illesztve

Object Constraint Language (OCL):
- Deklaratív (“Mit?” a “Hogyan?” helyett)
- programozási nyelvtől független
- UML-hez illeszkedik
- Lekérdez, nem változtat
- Kényszertípusok:
    - invariánsok
    - elő- és utófeltétel
    - metódustörzs
    - számított érték
    - kiindulási/alapértelmezett érték

XML: XML Metadata Interchange
- XML-alapú
- szabványos modell-leíró formátum

SOLID-elvek:

1. **Single Responsibility Principle**
- egy osztálynak egy jól meghatározható felelőssége legyen
- robosztus, tesztelhető, átlátható
- az erős kohézió és laza csatolást segíti elő

2. **Open/Closed Principle**
- a komponensek nyitottak legyenek a bővítésre, de zártak a módosításra

3. **Liskov Substitution Principle**
- objektumot lehessen a leszármazott osztályával kicserélni a helyes működés megsértése nélkül
- úgy írjuk a kódot, hogy a szabályoknak tényleg eleget tegyen pl. lista listaként is viselkedjen

4. **Interface Segregation Principle**
- ne függjön az osztály olyan interfészektől, amit az nem használ ki
- inkább specifikus, kicsi interfészek legyenek a nagyok helyett

5. **Dependency Inversion Principle**
- használjuk az interfészeket, absztrakt osztályokat a függőségeknek
- a függőségeket ne a függő hozza létre, hanem kívülről kapja

Tervezési elvek:
- Keep It Simple Stupid (KISS)
- You aren’t gonna need it (YAGI)
- Do not repeat yourself (DRY)

Összehasonlítós feladatok:

1. **Interfész vs. absztrakt osztály**

Interfész:
- csak előír metódust, nem implementál
- végrehajtani kell

Absztrakt osztály:
- örökölni kell
- átmenetet jelent interfész és osztály között
- némely metódust előírnak, némelyhez adnak implementálást
- közvetlenül nem példányosíthatók
- akkor használjuk, amikor több különböző osztályt akarunk közös jellemzőkkel ellátni és szükség szerint egy közös őssel referálni rájuk
- egy metódus absztrakt, ha az őt örkölő gyerekek közül legalább egy már másképpen akarja implementálni

Egy osztály nem örökölhet több, mint egy elvont osztályból, de több interfészt is képes megvalósítani.

2. **Kohézió vs. csatolás**

Kohézió:	Csatolás:
Egy komponensbe tartozó logikák mennyire szorosan függnek össze	Két komponensnek mennyire kell ismernie egymás belső működését
Erős/gyenge kohézió	Szoros/laza csatolás

Ideális: erős kohézió, laza csatolás

3. **ArrayList vs. LinkedList**

ArrayList	LinkedList
- tömböt használ - hatékony az indexelés és a végére fűzés - költséges a beszúrás, törlés	- láncolt listát használ - hatékony a beszúrás, törlés - költséges az indexelés

4. **IS-A vs. HAS-A relációk**

Öröklés: IS-A	Delegáció: HAS-A
- Modellezünk általános kategóriaként több dolgot és az általános működés öröklődik, ami kiegészíthető és felülírható	- Tagváltozóként tárolunk egy másik objektumot, és neki delegálunk hívásokat

User Case diagram: Include vs. Extend vs. Inheritance

- „Étteremben mindig fizetünk a végén” -> include
- „Fizetéskor néha adunk borravalót” -> extend
- „Két fajta étterméi fizetés van: kártyás és készpénzes” -> inheritance

Egyéb fogalmak, amikre rákérdezhetnek:

- **Sztereotípia:** meglévő elem jelentésének kiegészítése
    - <<interface>>
    - <<enumeration>>

  Dependency esetén:
    - <<use>>
    - <<create>>
    - <<destroy>>

- **Refactoring:** olyan átalakítás, amelyekkel a programkód működése nem változik
- **Dinamikus kötés:** futásidejű típus alapján dől el, hogyan viselkedik az objektum
- **Domén:** a szakterület, amelyhez a szoftver kapcsolódik
- **Clean code:** átlátható, jól kezelhető, bővíthető, karbantartható kód

1. **Template Method**

- Lehetővé teszi, hogy az algoritmus invariáns részeit egy helyen definiáljuk és a változó részeket a leszármazott osztályban adjuk meg.

- Utóbbi segítségével elkerülhető a kód duplikálás elkerülése (Do It Yourself elv):

  a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg, mely a különböző viselkedéseket megvalósító egyéb műveleteket hívja meg. Ezeket a műveleteket a leszármazott osztályban felül kell/lehet definiálni.

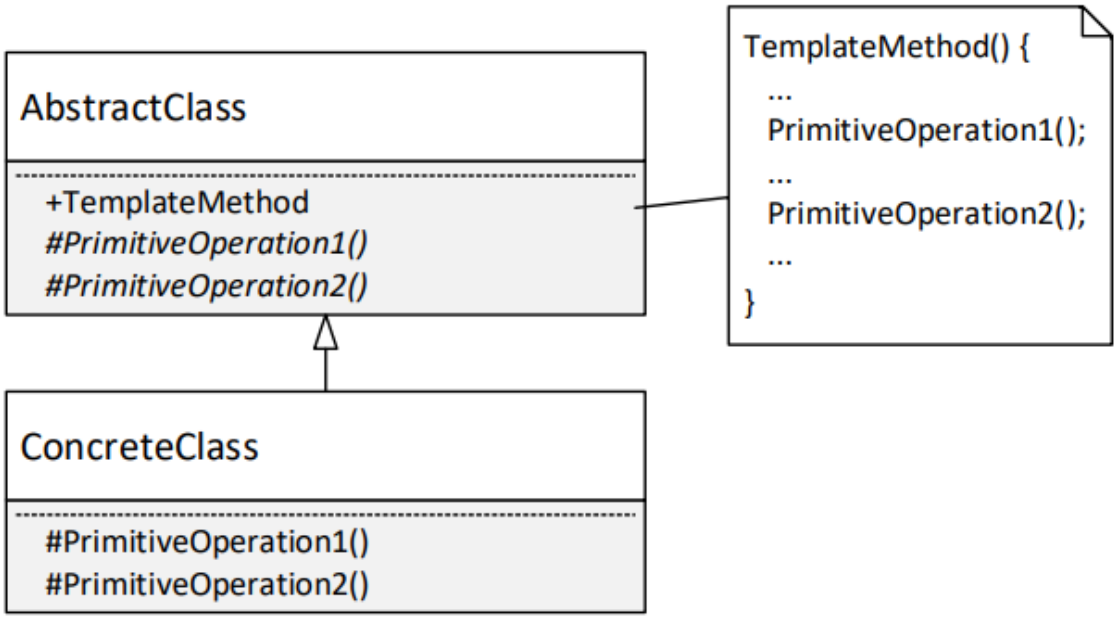
- Lehetővé teszi a kiterjesztési pontok definiálását (hook függvények)
- Kerestrendszerekben gyakran alkalmazzák

Pozitívum:

- Kódduplikációt elkerüljük
- Új viselkedésen könnyen bevezethető, nem kell a meglévő kódot változtatni

Negatívum:

- Futás közben nem tudjuk egyszerűen cserélni a viselkedést, emiatt rugalmatlan
- Sok keresztkombinációra lehet szükség, hiszen minden használt kombinációnak egy külön leszármazott osztály kell -> áttekinthetetlen, karbantarthatatlanná válik (ilyenkor általában Strategy mintára váltunk)



2. Strategy

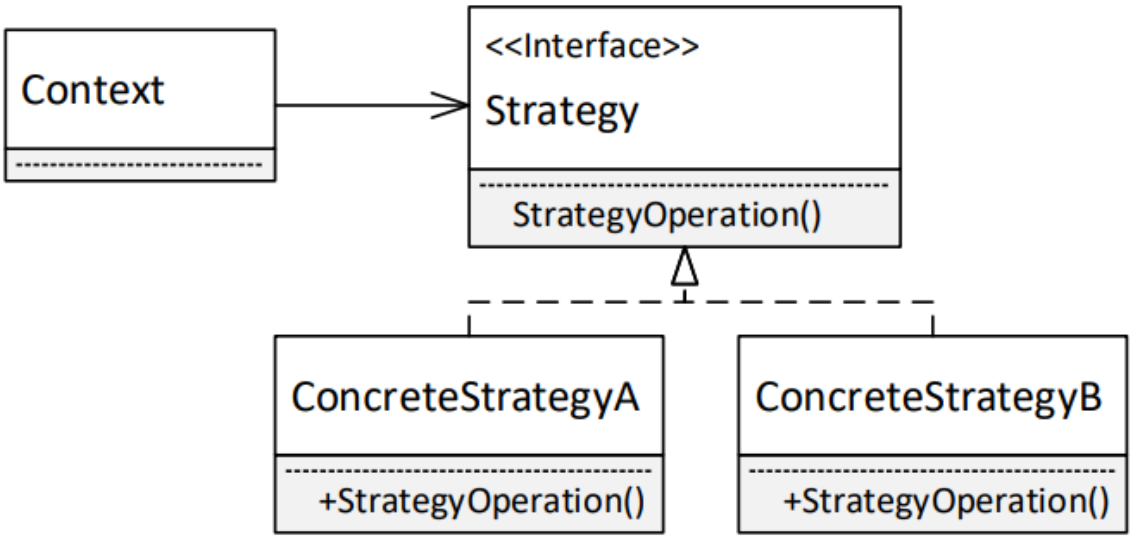
- Célja az algoritmusok/viselkedések egy csoportján belül az egyes algoritmusok/viselkedések egységbe zárása és egymással kicserélhetővé tétele (a kliens szemszögéből is).

- Minden olyan aspektusra, amit lecserélhetővé vagy bővíthető szeretnénk tenni, egy külön strategy hierarchiát vezetünk be

- Az osztály külön interface típusú hivatkozást tartalmaz minden aspektusra.

Pozitívum:

- Új viselkedés könnyen bevezethető, nem kell a meglévő kódot változtatni
- Több aspektus esetén nincs kombinatorikus robbanás az osztályhierarchiában
- Unit tesztelhetőségét segíti (a unit tesztek során az osztályt önmagában, a függőségei nélkül teszteljük)
- Program to an interface (and not to an implementation) elv egyik megtestesülése



3. Observer

- Célja:

  Lehetővé teszi, hogy egy objektum értesítést küldjön más objektumoknak az állapotának változásairól anélkül, hogy az alany függene a megfigyelők konkrét típusától.

  Használjuk, ha:
  - Amikor egy objektum megváltoztatása maga után vonja más objektum megváltoztatását, és nem tudjuk, hogy hány objektumról van szó
  - Amikor egy objektumnak értesítenie kell más objektumokat az értesítendő objektumot szerkezetére vonatkozó feltételezések nélkül

x

- Könnyen bővíthető a laza csatolás megvalósításával

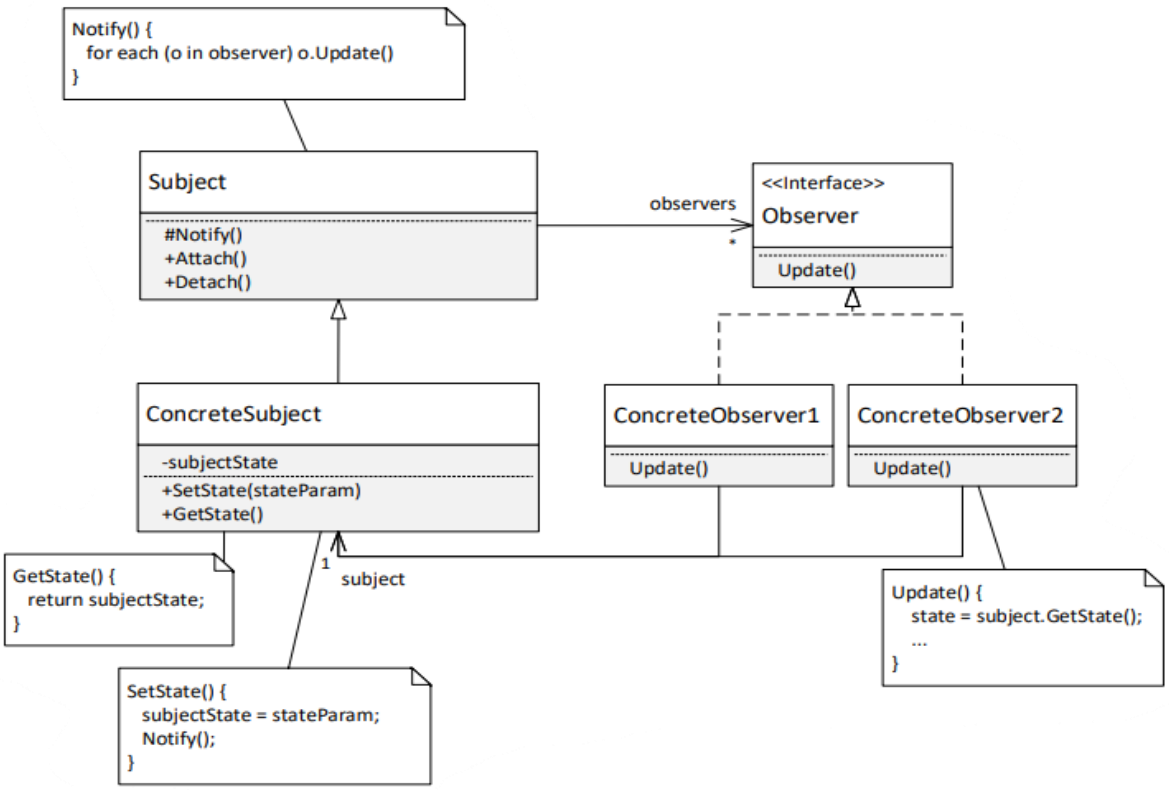
Szereplők:

1. Subject
   - Lehetőséget ad az Observereknek a be- és kiregisztrálására, értesítésére. Ezáltal tartalmazza a különböző ConcreteObserver-ek közös kódját (közös ős).
   - Tárolja a beregisztrált Observereket
   - A gyakorlatban nem jelenik meg mindig.

2. Observer
   - Interfészt definiál azon objektumok számára, amelyek értesülni szeretnének a Subject-ben bekövetkezett változásról (Update művelet)

3. ConcreteSubject
   - Az Observer-ek számára fontos állapotot tárol
   - Értesítési a beregisztrált Observer-eket, amikor az állapota megváltozik

4. ConcreteObserver
   - Referenciát tárol a megfigyelt ConcreteSubject objektumra
   - Olyan állapotot tárol, amit a megfigyelt ConcreteSubject állapotával konzisztensen kell tartani
   - Implementálja az Observer interfészét



4. Dependency Injection

- Nincs a GoF könyvben, de logikailag passzol

A függőség megfordítása, az egyértelmű felelősség elvét következtetve lazít az objektumok közötti csatoláson azzal, hogy elkülöníti a létrehozást és a használatot.
Alapegysége az injekció, hasonló a paraméterátadáshoz.

- A klienst elszigeteli a részletektől
- Az átadás független a klienstől és attól hogyan valósul meg

Elemei:

- Kliens: egy objektum, ami igénybe vesz egy másik objektumot.
- Szolgáltató objektum: szolgáltatást nyújt egy kliens részére.
- Interfész: a kliens által elvárt típus.
- Injektor: odaadja a szolgáltató objektumot a kliensnek, beállítja, injektálja.

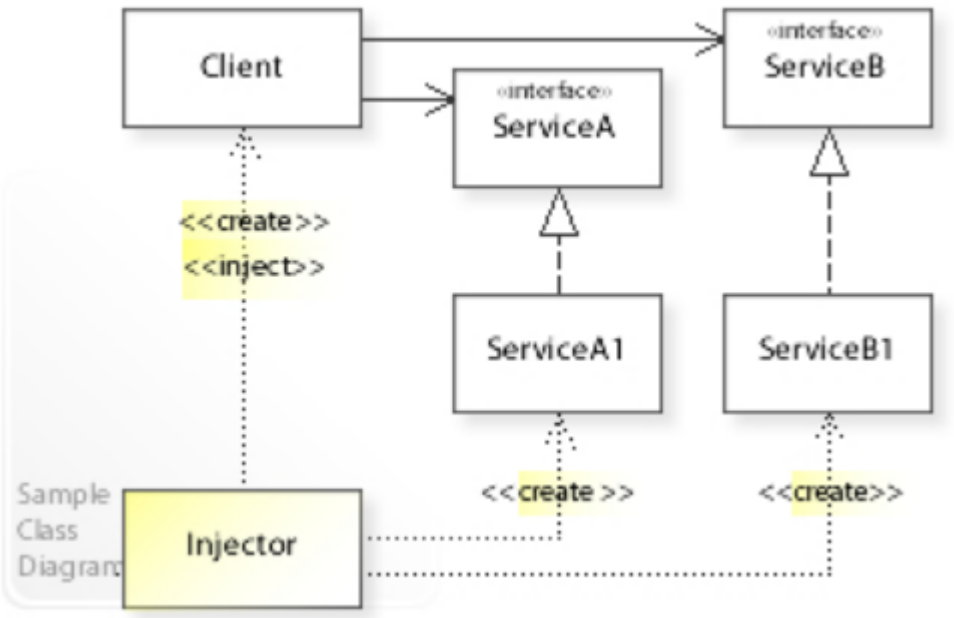
- A kliens csak azt láthatja, amit az interfész előír
- A kliens nem ismerheti a szolgáltató objektum konkrét megvalósítását
- A szolgáltató objektum változásai nem érintik a kliens addig amíg az interfész nem változik

Előnyök:

- A kliens bármivel működik, ami támogatja a kliens által elvárt interfészt -> rugalmas
- Egyszerű unit teszteket végezni, mert a kliensek függetlenebbek
- Támogatja a karbantarthatóság és újrafelhasználhatóságot, mert a kliensből eltávolítanak minden olyan információt, ami az általa használt objektumok megvalósításának bármely részletét is tartalmazza
- Boilerplate code csökkenése, mert az inicializálást és beállítást külön komponens végzi

Hátrányai:

- Több fájlban kell végig követni a rendszer viselkedését -> hibalehetőség, nehéz megtalálni a hiba helyét is
- Arra készteti a fejlesztőket, hogy függjenek egy keretrendszertől
- A minta a fejlesztés irányát és sorrendjét is megszabja

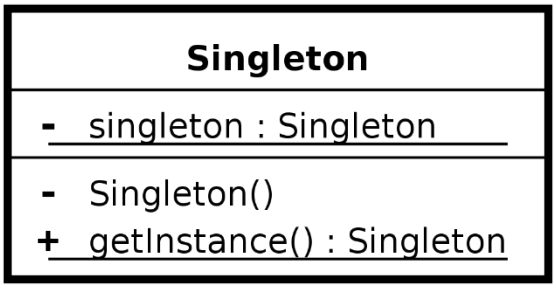


5. **Singleton**

    - Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni és ehhez az egy példányhoz globális hozzáférést biztosít
    - Pl. Központi ablakkezelő
    - Használatát nem szabad túlzásba vinni, antipattern

Ezt rendszerint így érik el:

- minden konstruktor privát
- a példány referenciája egy osztálymetóduson keresztül érhető el



6. Abstract Factory

Célja: Interfész biztosít ahhoz, hogy egymással összefüggő objektumot családjait hozzuk létre anélkül, hogy specifikálnánk a konkrét osztályaikat.

Használjuk, amikor:

- A rendszernek függetlennek kell lennie az általa létrehozott dolgoktól (“termék” objektumok, pl. felhasználói felület elemek)
- A rendszernek több termékcsaláddal kell együttműködnie
- A rendszernek szorosan összetartozó “termék” objektumok adott családjával kell dolgoznia, és ezt akarjuk kényszeríteni a rendszerben (pl. Win10 scrollbart ne lehessen OSX ablakkal együtt használni)

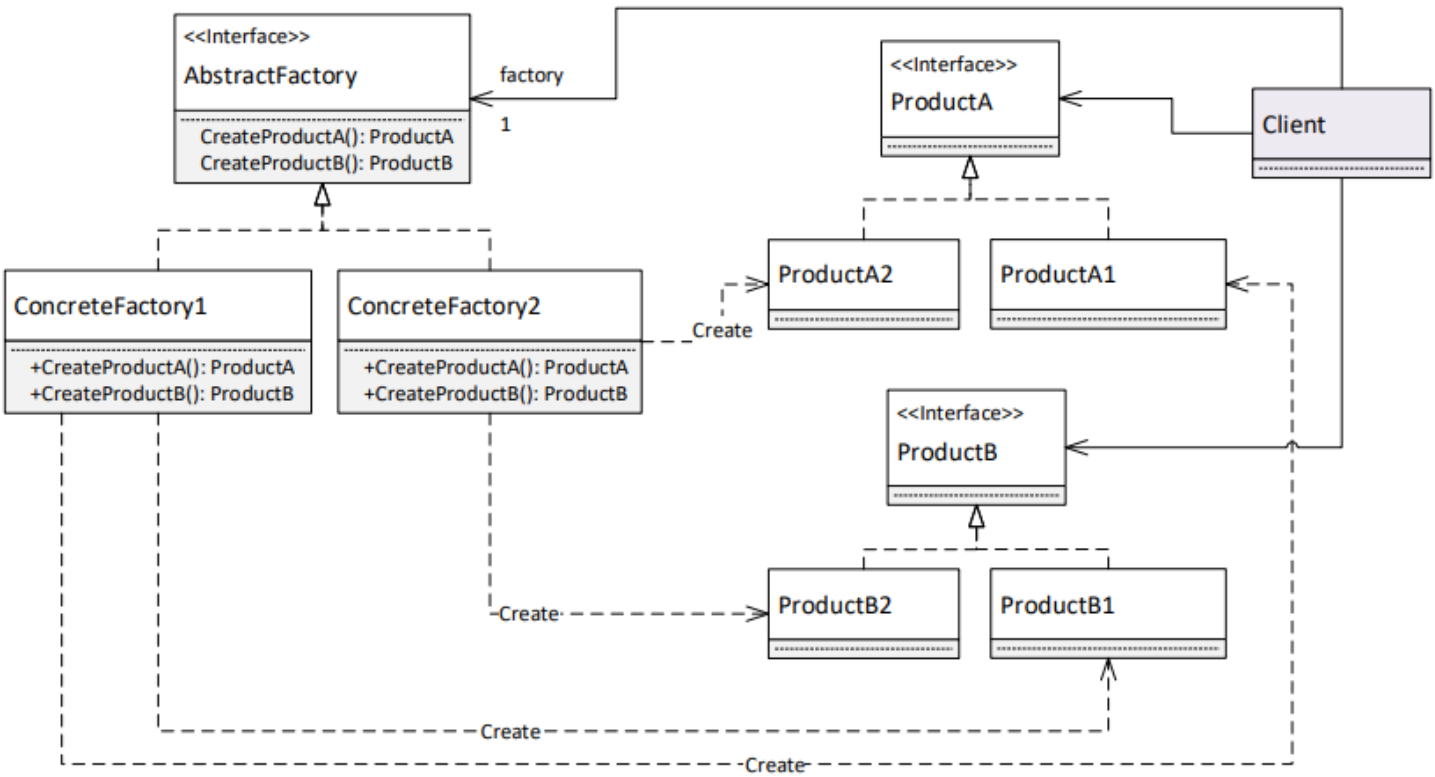
Előnyök:

- Elszigeteli a konkrét osztályokat
- A termékcsaládokat könnyű kicserélni
- Elősegíti a termékek közötti konzisztenciát

Hátrányok:

- Nehéz új termék hozzáadása. Ekkor az Abstract Factory egész hierarchiáját módosítani kell, mert az interfész rögzíti a létrehozható termékeket.

Megjegyzés: ezt bizonyos esetekben ki lehet kerülni, pl. ha minden termék egy közös osztályból származik)



GoF tervezési minták csoportosítása:

Létrehozási (creational):

- *Abstract Factory*
- *Singleton*

Viselkedési (behavioral):

- *Template Method*
- *Strategy*
- *Observer*