Szoftvertechnológia és -technikák

7. Előadás – Benedek Zoltán

Tervezési minták 2

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

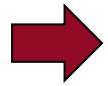
Tervezési minták 2

- Observer
- Létrehozási minták
  - Dependency Injection
  - Singleton
  - Abstract factory
  - (Factory method – nem tananyag)

Observer (Megfigyelő)

Observer célja

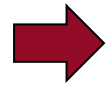
- Cél (két megközelítésben)
  - Lehetővé teszi, hogy egy objektum (subject/alany) értesítést küldjön más objektumoknak (observer/megfigyelő) az állapotának változásairól. Mindezt anélkül, hogy az alany függene a megfigyelők konkrét típusától.
  - Lehetővé teszi, hogy objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne köztük a konkrét osztályaiktól (ez az absztraktabb definíció, mélyebb célokat fogalmaz meg).

Még nem értjük, lássunk egy példát 

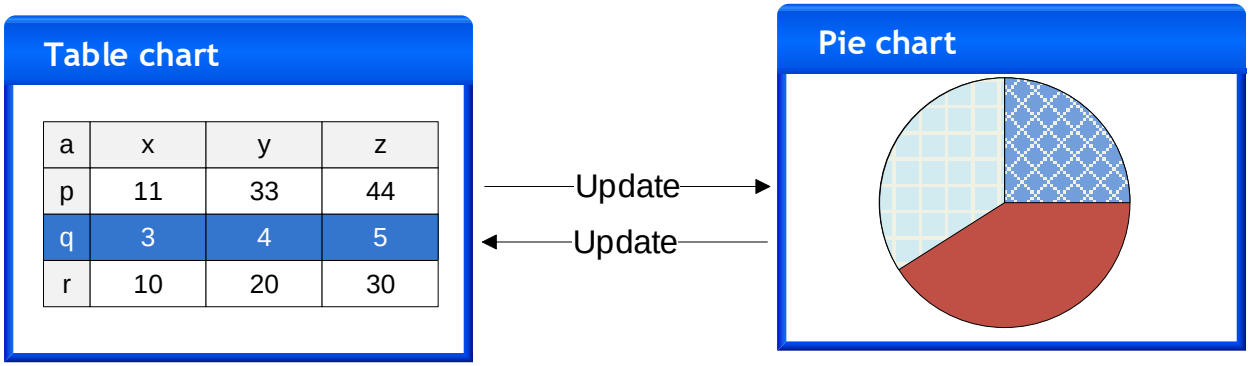
Példa

- Egy táblázatkezelő alkalmazásban támogatni kell az adatok különböző megjelenítését
  - Első körben **TableView** és **PieChartView**
  - T.f.h. a felhasználó mindkettőben tudja módosítani az adatokat
  - Meg kell oldani, hogy a két nézet konzisztens nézetét mutassa az adatoknak. Bármelyikben módosul az adat, azt jeleznie kell a másiknak.
    - Megoldás első körben: mindkét osztály tartalmaz egy hivatkozást a másikra

Ábra és kód



Példa



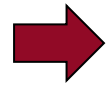

```
class TableView
{
    // ...

    double[,] data;
    PieChartView pieChartView;

    public void Update(double[,] data) {
      /* this.data beállítása data param alapján */
    }
    public void SetCellData(int col, int row, double value)
    {
        this.data[col, row] = value;
        pieChartView.Update(data);
    }
    public void Draw() { /* ... */ }
   }
```

A **PieChartView** kódja hasonló, csak a **TableView**-ra tartalmaz hivatkozást.

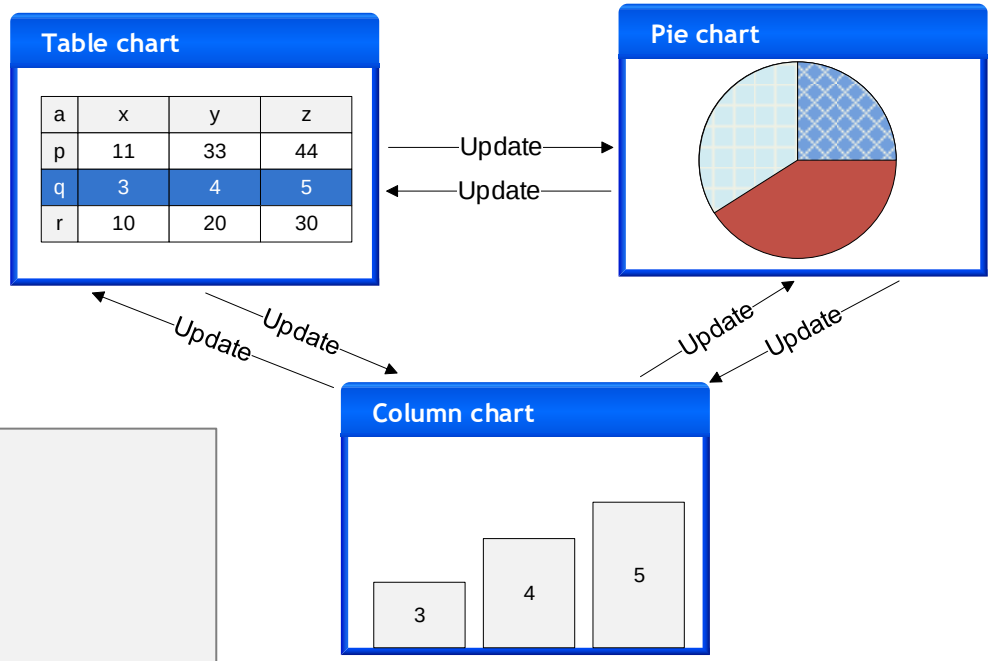
Példa

- T.f.h bővíteni kell a megoldásunkat
  - Be kell vezetni az oszlopdiagramos megjelenítést (**ColumnChartView**)
  - T.f.h. ebben is lehet módosítani az adatokat
  - Konzisztens nézetek: bármelyik nézet módosul, frissíteni kell a másik kettőt is!
  - Új nézetet szeretnék bevezetni, minden MEGLÉVŐ nézet osztályt módosítani kell ☹☹☹
    - Fel kell vegyük a meglévő osztályokba a hivatkozást az új nézetre

Ábra és kód



Példa



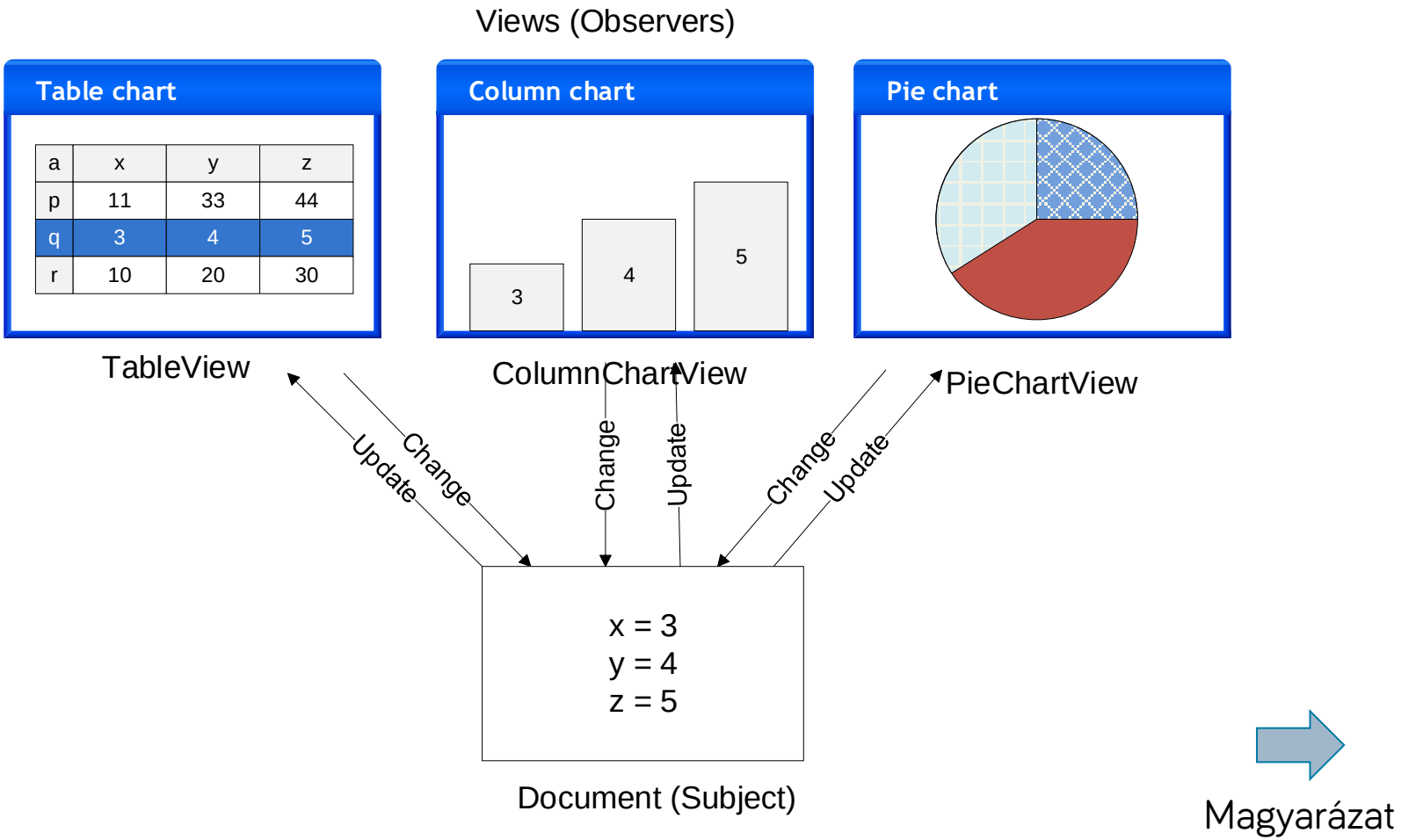

```
class TableView
{
    // ...

    double[,] data;
    PieChartView pieChartView;
    ColumnChartView columnChartView;

    public void Update(double[,] data) {
        /* this.data beállítása data param alapján */
    }
    public void SetCellData(int col, int row,
       double value)
    {
        this.data[col, row] = value;
        pieChartView.Update(data);
        columnChartView.Update(data);
    }
    public void Draw() { /* ... */ }
   }
```

A **PieChartView** és a **ColumnChartView** kódja hasonló: a másik két nézetre tartalmaz hivatkozást

Példa kiértékelése

- A megoldásunk (mindenki-mindenkivel összekötve, közvetlen függvényhívással) eddig is nehezen volt bővíthető
  - A további bővítés teljes rémálom
  - Áttekinthetetlen + új nézet bevezetésekor mindig módosítani kell a meglévő nézeteket is
  - Nehéz karbantartani, továbbfejleszteni, újra felhasználni részeket, mert túl szoros a csatolás az osztályok között (a nézetek függenek egymástól).
- Alkalmazzuk a példánkban az **Observer** mintát

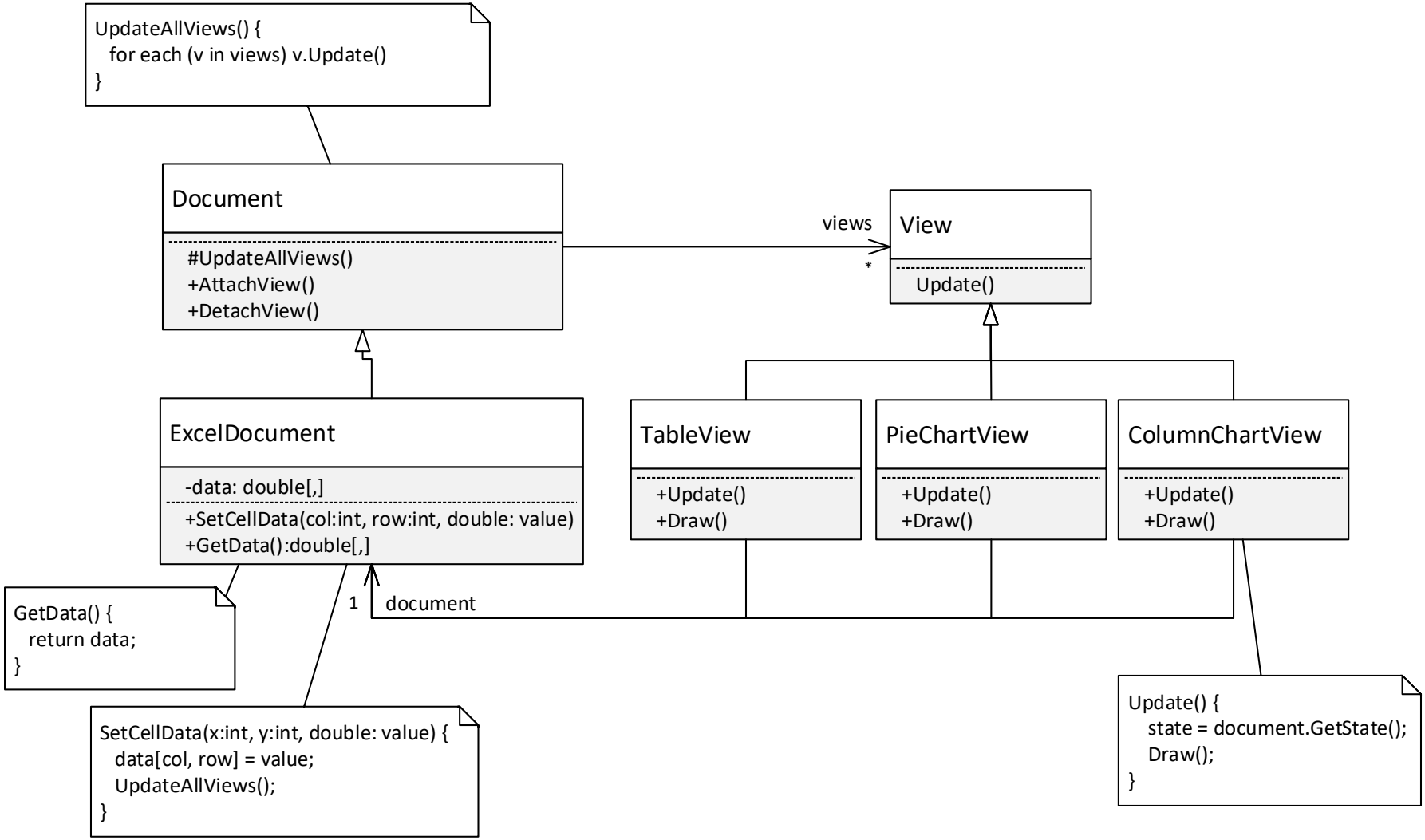
Nézetek frissítése Observer mintával



Nézetek frissítése Observer mintával

- Magyarázat, minden pont fontos!
  - Az adatoknak egyetlen, közös „hiteles” forrása legyen: emeljük ki az adatokat és az azon értelmezett műveleteket egy osztályba, ez lesz a **dokumentum** (subject)
  - A dokumentumhoz különböző **view**-kat lehet beregisztrálni (observer)
  - Ha valamelyik view megváltoztatja a dokumentum adatait, a dokumentum értesíti az összes beregisztrált view-t a változásról
  - Az értesítés hatására a view-k lekérdezik a dokumentum állapotát és frissíti magát
  - A dokumentum csak egy közös View interfészen/ősosztályon keresztül tárolja a beregisztrált view-kat (nem függ az egyes nézet típusoktól)

Példa, osztálydiagram



Osztálydiagram magyarázat

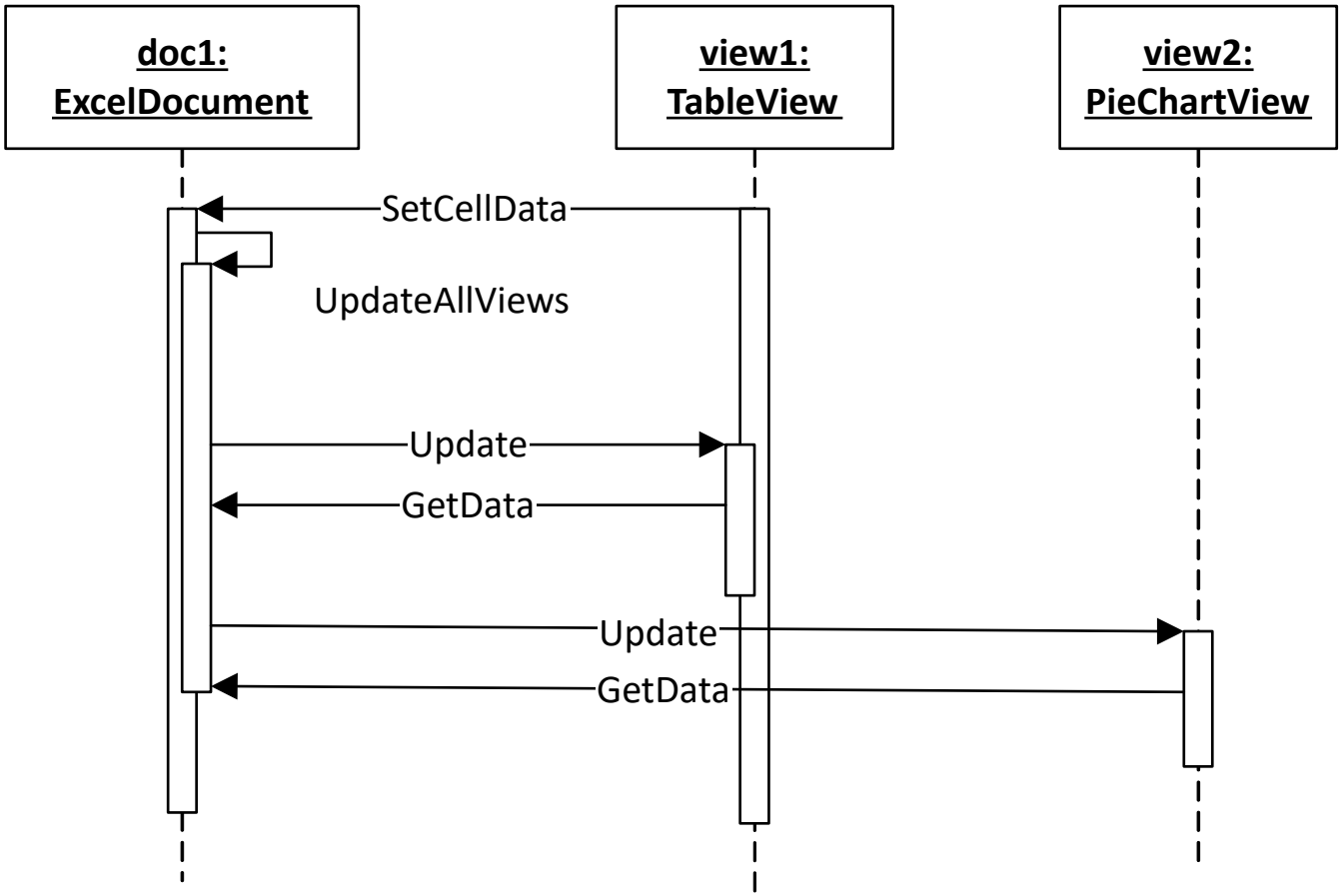
- Document
  - Az osztály objektumai reprezentálnak egy-egy dokumentumot, és a *views* nevű listájában tárolja a beregisztrált nézeteket.
  - Tartalmazza a dokumentumokra közös kódot, ha több különböző típusú dokumentum is van (a példánkban nincs)
- ExcelDocument
  - Egy példa *Document* leszármazottra.
  - Tárolja a *data*, stb. tagváltozójában a dokumentum adatait (pl. cella értékek).
  - Publikus lekérdező függvényeket biztosít a többi osztály számára az adatokhoz (elsősorban a nézet számára), pl. *GetData*.
  - Publikus adatmódosító műveleteket biztosít a többi osztály számára, pl. *SetData*. Ezek módosítják a tagváltozókat, majd az *UpdateAllViews* hívásával értesítik a többi nézetet a változásról.
  - Az *UpdateAllViews* frissíti a beregisztrált nézeteket (Update-et hív mindre).

Osztálydiagram magyarázat

- View
  - Az egyes nézetek közös őse vagy interfésze, lehetővé teszi egységes kezelésüket.
- TableView, PieChartView, stb.
  - A dokumentum egyes nézeteit reprezentálják.
  - A *View-ból* származnak (vagy a View interfészt implementálják, ha a *View* interfész)
  - Felüldefiniálják vagy implementálják a *View Update* műveletét. Az *Update* műveletben a nézet a dokumentum aktuális állapota alapján frissíti magát.
  - Tartalmaznak egy referenciát a dokumentumra, amin keresztül a leszármazott nézetek elérhetik a dokumentumot, melynek adatait megjelenítik.
    - Alternatív megoldás lehet, ha a View közös ős tartalmazza ezt a referenciát a *Document* ősre
- Nézetben hivatkozás a *Document* (subject)-re, két lehetőség
  - A példában a *View* implementációkba tettük, feltételezve, hogy a View egy interfész
  - Ha a *View* osztály (nem interfész), akkor a *View* ősbe is tehető.
    - A gyakorlatban a *View*-kat a .Net, Java, stb. keretrendszer Form/Window/Control/egyéb osztályából származtatjuk

Példa szekvenciadiagram

- Pár diával korábban szövegesen már szerepeltek a lépések



Példa kód

- Kód: lásd *ObserverDocView* mappa
- Egy kicsit szofisztikáltabb változat:
  - A nézet vonatkozásában van *IView* interfész, de egy *ViewBase* közös ősosztály is: ez utóbbiba a nézetekre közös kód került.
- Az egyszerűség kedvéért egy ez konzol alkalmazás, a *ColumnChartView* és a *PieChartView* így természetesen nem rajzol

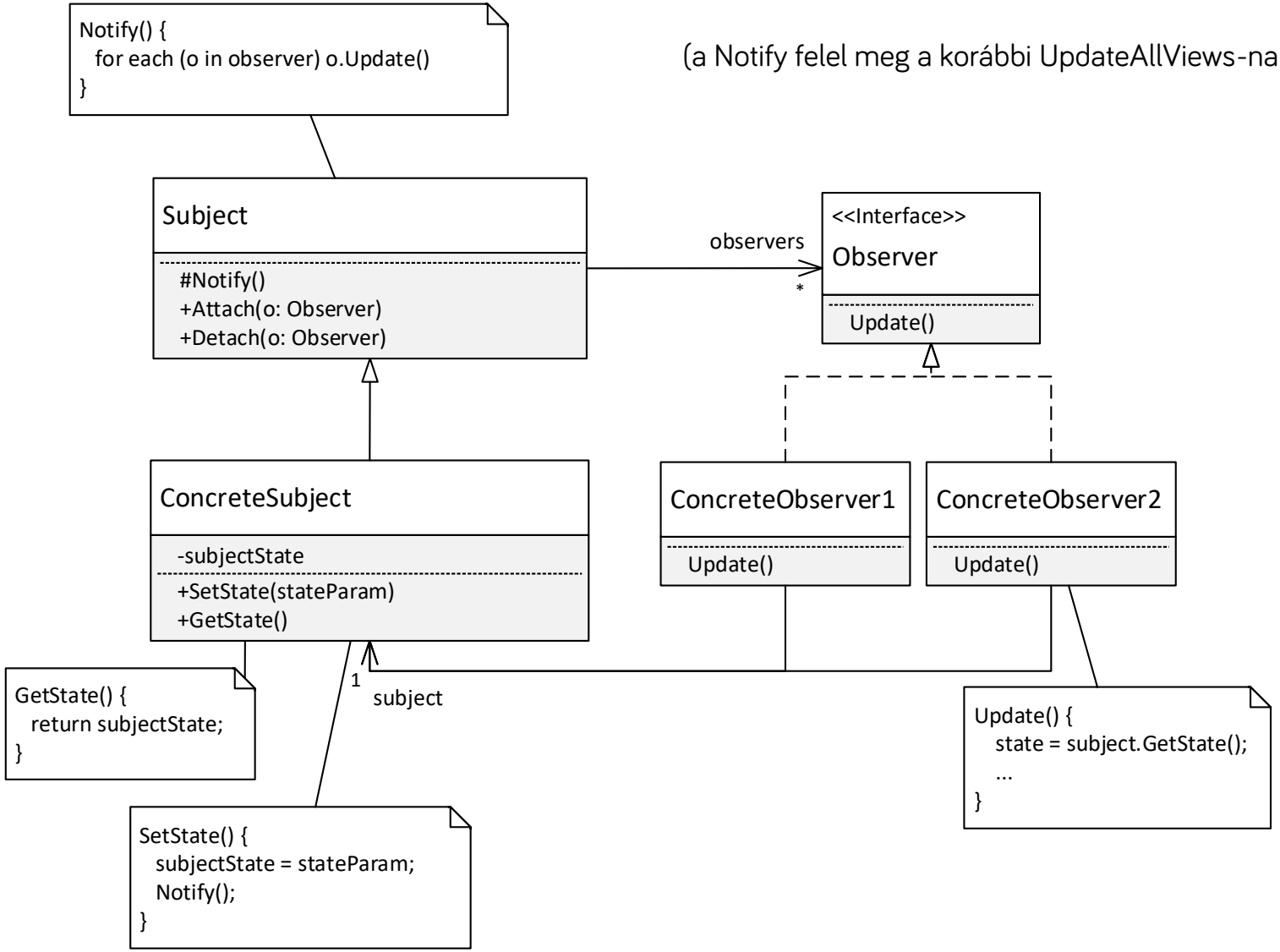
Példa összefoglaló

- Ez egy példa volt, az Observer minta alkalmazására egy speciális esetre, adatok megjelenítésére
  - Ez maga a Document-View architektúra, még visszatérünk rá pár előadás múlva
- Observer általánosítva
  - Lehetővé teszi, hogy egy objektum (subject/alany) értesítést küldjön más objektumoknak (observer/megfigyelő) az állapotának változásairól. Mindezt anélkül, hogy az alany függene a megfigyelők konkrét típusától.
- További példák gyakorlaton

Observer

- Előnyök
  - A rendszer könnyen kiterjeszthető új view osztályokkal. Sem a document (subject), sem a többi view (observer) osztályt nem kell ehhez módosítani.
    - Próbáljuk ki, vezessünk be egy új nézet osztályt!
  - Egy egyszerű mechanizmust kaptunk arra, hogy az összes view konzisztens nézetét mutassa az adatoknak.
  - A document (subject) kódjában csak egy *View* (observer) lista van, így a modell független az egyes *View-t* implementáló osztályoktól. A document (subject) újrafelhasználható!
- Általánosítva, elvonatkoztatva a document-view esettől
  - A fenti megközelítés lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk
  - Ez az ún. *observer* minta lényege

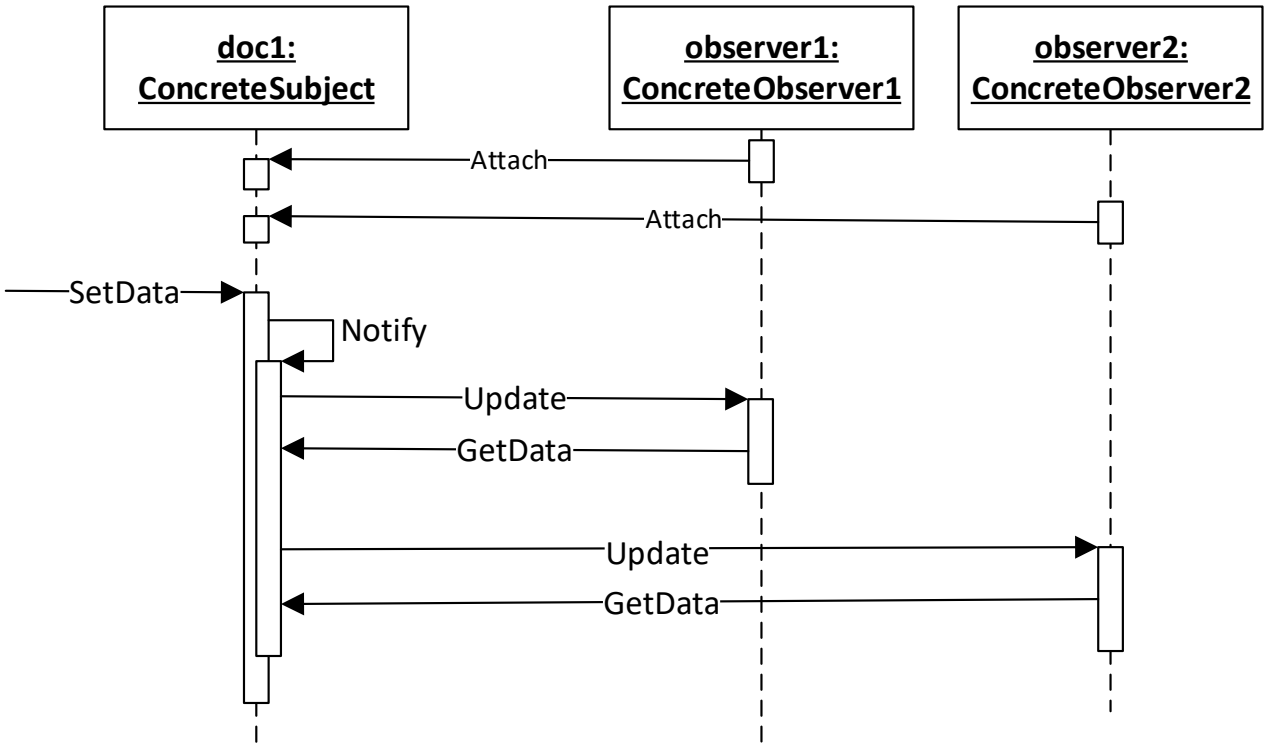
Observer osztálydiagram



Observer - szereplők

- Subject
  - Tárolja a beregisztrált Observer-eket
  - Lehetőséget ad Observer-ek be- és kiregisztrálására valamint értesítésére. Ezáltal „tartalmazza” a különböző ConcreteObserver-ek közös kódját (közös ős).
  - A gyakorlatban nem jelenik meg mindig
- Observer
  - Interfészt definiál azon objektumok számára, amelyek értesülni szeretnének a Subject-ben bekövetkezett változásról (Update művelet)
- ConcreteSubject
  - Az observer-ek számára érdekes állapotot tárol
  - Értesíti a beregisztrált Observer-eket, amikor az állapota megváltozik
- ConcreteObserver
  - Referenciát tárol a megfigyelt ConcreteSubject objektumra
  - Olyan állapotot tárol, amit a megfigyelt ConcreteSubject állapotával konzisztensen kell tartani
  - Implementálja az Observer interfészét (Update művelet), ez az, amit a Subject meghív, amikor a ConcreteSubject állapota megváltozik. Ebben frissíti a saját állapotát a megfigyelt ConcreteSubject állapotának megfelelően

Observer dinamikus nézet



- A subject állapotváltozását (SetData hívás) nem csak observer válthatja ki! Ezért az ábrán ezt egy kívülről jövő művelettel jelöltük.
- Megjegyzés: „Sajnos” az UML szekvencia diagram csak objektumokat ábrázol, nem képes kifejezni, hogy milyen interfészen keresztül hivatkoznak egymásra az objektumok, így a függetlenséget nem tudja kifejezni

Observer

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

Observer – záró gondolatok

- Gyakran használt minta
- Laza csatolás megvalósításával segíti a könnyű bővíthetőséget
  - Az observerek nem direktben kommunikálnak egymással, egymásról nem is tudnak, csak a subject-et ismerik
  - A subject nem függ az egyes observerek konkrét típusától (csak egy observer interfésztől)
  - Új observer bevezetésekor nem kell módosítani sem a subject-et, sem a többi observert, csak implementálni kell az observer interfészt
- A .NET esemény (event) is ennek egy speciális megjelenése: egy osztály minden event tagja egy „subject”, amire a += operátorral iratkoznak fel az observerek

Observer – záró gondolatok

- A példáinkban változás esetén a subject nem adja át az observernek az adatot. Csak a változás tényét jelzi, az observer egy plusz lépésben kéri le az adatot.
  - Teljesen érvényes megoldás lehet, ha a subject elküldi a változott adatot, paraméterben átadva az observernek (különösen, ha kisebb méretű az adat)
  - Esetfüggő, melyik megközelítés a praktikusabb

LÉTREHOZÁSI MINTÁK

Dependency Injection (DI)

Abstract factory

Singleton

(Factory Mehod – nem tananyag)

Létrehozási minták

- Objektumok létrehozásával kapcsolatos
- A new-val való közvetlen objektum létrehozás sokszor rugalmatlan
  - Emlékezzünk a korábbi alapelvünkre: zárjuk egységbe a kód azon részeit, melyeknél arra számítunk, változni/bővülni fognak
  - Ezek a részek gyakran az objektumok létrehozásával kapcsolatosak
    - Pl. nem akarjuk beégetni, milyen típusú objektumot hozzon létre egy adott kódsor (a new operátor esetében ez be van égetve, ez maga a típus, mely a new után áll, pl. *new TextDocument()* esetén TextDocument típus)
  - Rövidesen nézünk példákat …

Létrehozási minták

- Megoldási lehetőségek
  - Dependency Injection -DI
  - Factory Method
  - Abstract Factory
  - Singleton – ez egy kicsit speciális célokat szolgál
  - ...

Dependency Injection, DI (függőséginjektálás)

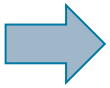
Dependency Injection

- Ez nem egy klasszikus tervezési minta, nincs a GoF könyvben, de logikailag ide passzol
- Egy példán keresztül világítjuk meg
  - A célunk egy olyan **`SecurityService`** osztály megírása, mely felhasználókhoz kapcsolódó biztonsági szolgáltatásokat biztosít, úgymint jelszó megváltoztatása, login, stb.
  - A felhasználókat valamilyen adatbázisban tároljuk.

SecurityService példa – kezdeti verzió

- Kód: lásd DesPattCode.DepencencyInjection.Initial mappa

```
class SecurityService
{
    // A userId azonosítójú felhasználó jelszavát megváltoztatja
    // newPassword-re (ha teljesíti a validációs szabályokat)
    // Visszatérési értékben jelzi, hogy sikerült-e a jelszó megfelel-e
    // a validálási szabályoknak.
    public bool ChangePassword(int userId, string newPassword)
    {
        var userRepository = new UserRepository();
        // Adatbázisból régi jelszó lekérdezése userRepository segítségével
        var oldPassword = userRepository.GetUserPassword(userId);

        // Jelszó validálás
        bool isValid = isPasswordValid(newPassword, oldPassword);
        if (!isValid)
            return false;

        // Adatbázisból jelszó módosítása userRepository segítségével
        userRepository.UpdateUserPassword(userId, newPassword);

        return true; // ha minden stimmel
    }
    …
}
```

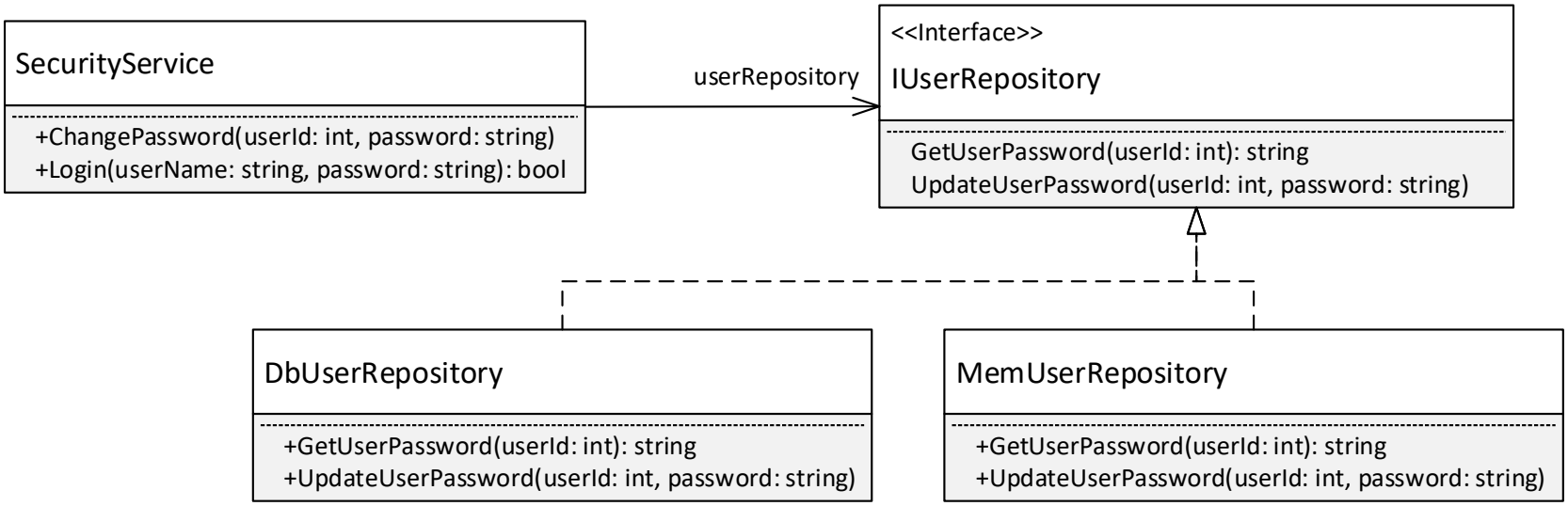


Magyarázat

SecurityService példa – kezdeti verzió

- Arra figyeltünk, hogy a felhasználók adatainak lekérdezése, módosítása (perzisztencia) le van választva egy **`UserRepository`** osztályba, hiszen az egy másik felelősségi kör (SRP elv).
- Probléma1: a **`ChangePassword`**-be a **`UserRepository`** osztály alkalmazása be van égetve, csak ezzel tud dolgozni. Rugalmatlan, csak adott adatbázis alapú felhasználó tárolással tud dolgozni. Ha más implementációt szeretnénk, a `var` **`userRepository = new UserRepository()`** sorokat mind át kell írni, ráadásul számos helyen.
- Probléma2: Az előző problémából adódóan a **`ChangePassword`** nem unit tesztelhető. A **`ChangePassword`** tesztelésénén NEM akarjuk tesztelni az adatbázis alapú perzisztenciát végző kódot, hanem csak a **`SecurityService`-ben** levő logikát. Ezen túl az adatbázis alapú perzisztencia nagyban feleslegesen lassítja a unit teszt futását.
- Megoldás: alkalmazzuk a Strategy patternt, vezessünk be egy **`IUserRepository`** interfészt több lehetséges implementációval (adatbázis alapú az éles működéshez, memória alapú a unit teszteléshez), a **`SecurityService`** pedig interfészként hivatkozzon rá. Lásd WithStrategy mappa...

SecurityService példa – Strategy-vel

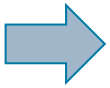


- Csak alkalmaztuk/gyakoroltuk a Strategy mintát
  - **`IUserRepository`** – interfész
  - **`DbUserRepository`** – egy implementáció, mely adatbázisba dolgozik (éles környezethez)
  - **`MemUserRepository`** – egy implementáció, mely a memóriába dolgozik (így gyors) unit teszteléshez

SecurityService példa – Strategy-vel

- Lásd DesPattCode.DepencencyInjection.WithStrategy mappa

```
class SecurityService
{
    // Interfészként hivatkozunk a user repository implementációra
    IUserRepository userRepository;

    // Egy központi helyen, a konstrukorban példányosít egy implementációt
    public SecurityService()  {  userRepository = new DbUserRepository(); }

    // A userId azonosítójú felhasználó jelszavát megváltoztatja
    // newPassword-re (ha teljesíti a validációs szabályokat)
    // Visszatérési értékben jelzi, hogy sikerült-e a jelszó megfelel-e
    // a validálási szabályoknak.
    public bool ChangePassword(int userId, string newPassword)
    {
        // Adatbázisból régi jelszó lekérdezése userRepository segítségével
        var oldPassword = userRepository.GetUserPassword(userId);

        // Jelszó validálás
        bool isValid = isPasswordValid(newPassword, oldPassword);
        if (!isValid)
            return false;

        // Adatbázisból jelszó módosítása userRepository segítségével
        userRepository.UpdateUserPassword(userId, newPassword);

        return true; // ha minden stimmel
    }
   …
}
```



Magyarázat

SecurityService példa – Strategy-vel

- Az eredeti(Initial) **`SecurityService`** továbbfejlesztése.
- Interfészként hivatkozik a **`UserRepository`** implementációra: Ha más implementációt szeretnénk, akkor a műveleteket, pl. a **`ChangePassword`**-öt nem kell megváltoztatni:
- Még mindig maradt egy probléma, a konstruktorban a new-val a példányosítás! A new után csak konkrét implementációt adhatunk meg. így ha át szeretnénk térni egy másik implementációra, akkor a SecurityService kódját még mindig módosítani kell (igaz, már csak egy helyen, a konstruktorban), ez sérti az Open/Closed elvet (meg kell nyitni az osztályt módosításra).
- Megoldás: Dependency Injection(DI) alkalmazása, konstruktor, vagy esetleg metódus paraméterben adjuk át az implementációt a **`SecurityService`** osztálynak. Lásd WithStrategyAndDI mappa.

Dependency Injection (DI) alkalmazása a Strategy mellett)

- Lásd DesPattCode.DepencencyInjection.WithStrategyAndDI mappa

```
class SecurityService
{
    // Interfészként hivatkozunk a user repository implementációra
    IUserRepository userRepository;

    // Konstruktor paraméterként kap egy  IUserRepository implementációt
    public SecurityService(IUserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // A userId azonosítójú felhasználó jelszavát megváltoztatja
    // newPassword-re (ha teljesíti a validációs szabályokat)
    // Visszatérési értékben jelzi, hogy sikerült-e a jelszó megfelel-e
    // a validálási szabályoknak.
    public bool ChangePassword(int userId, string newPassword)
    {
        // Adatbázisból régi jelszó lekérdezése userRepository segítségével
        var oldPassword = userRepository.GetUserPassword(userId);

        // Jelszó validálás
        bool isValid = isPasswordValid(newPassword, oldPassword);
        if (!isValid) return false;

        // Adatbázisból jelszó módosítása userRepository segítségével
        userRepository.UpdateUserPassword(userId, newPassword);

        return true; // ha minden stimmel
    }
```

Dependency Injection (DI) alkalmazása a Strategy mellett)

- Folytatás, **`DemoSecurityService`** osztályban **`SecurityService`** használatra példák:

```
// Éles alkalmazásban a DbUserRepository-val használjuk a SecurityService-t
var securityService = new SecurityService( new DbUserRepository() );
securityService.ChangePassword(111, "abrakadabra");


// Unit teszt esetében a gyors MemUserRepository-val használjuk a
// SecurityService-t
var securityService2 = new SecurityService( new MemUserRepository() );
// ... teszt előkészítés ...
// tesztelendő kód futtatás
bool wasValid = securityService2.ChangePassword(111, "abrakadabra");
//  ... teszt validálás ..
```

Magyarázat

Minden pont fontos!

- A **`SecurityService`** Interfészként hivatkozik a **`UserRepository`** implementációra, és az implementációt is kívülről kapja meg (dependency injection) jelen példában konstruktor paraméterként.
- Minden korábbi problémától megszabadultunk, sehol nem szerepel a **`SecurityService`** osztályban egyik **`IUserRepository`** implementáció sem (próbáljuk rákeresni a fájlban a DbUserRepository és MemUserRepository szövegekre – nem lesz találat).
- A **`SecurityService-et`** így bármelyik **`IUserRepository`** implementációval is akarjuk használni, a **`SecurityService`** kódját nem kell módosítani (lásd **`DemoSecurityService`** osztály)

További DI gondolatok

- A DI-t már előző előadáson is ösztönösen használtuk, amikor a **`DataProcessor`** osztálynak a **`ICompressionStrategy`** és **`ICancellationStrategy`** implementációkat adtunk át!
- Általában a konstruktor paraméterként való megadást preferáljuk, de ha futás közben szükség van az implementáció megváltoztatására, akkor egy setter metódust használunk, pl. a **`SecurityService`**-ben.:

```
public void SetUserRepository(IUserRepository
   userRepository)
{
    this.userRepository = userRepository;
}
```

- A gyakorlatban sokszor a keretrendszer extra szolgáltatásokat (ún. DI containert) nyújt ahhoz, hogy a DI-t igazán kényelmesen tudjuk használni, pl. .NET, Java Spring. Ez a tárgy keretében nem szerepel.

További DI gondolatok

- Bizonyos esetekben a DI nem használható, mert a feladat olyan jellegű, hogy az osztálynak kell adott pontban – még ha közvetve is – példányosítani az objektumokat.
- Ez esetben a Factory Method és az Abstract Factory minták használhatók
  - Illetve a Prototype és a Builder minták, de ezeket nem tanuljuk

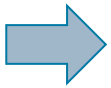
Singleton (Egyke)

Singleton

- Célja
  - Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez az egy példányhoz globális hozzáférést biztosít
- Globális hozzáférés
  - Az egy példány kódban bárhonnan kényelmesen elérhető, anélkül, hogy függvényparaméterben kellene átpasszolni
- Példa
  - Pl. egy központi ablakkezelő (pl. WindowManager) vagy Application objektum
- Megoldás
  - Singleton
    - Legyen az osztály felelőssége, hogy csak egy példányt lehessen belőle létrehozni
    - Maga az osztály biztosítson globális hozzáférést ehhez az egy példányhoz
  - A programozási nyelvek segítségét igényli

Singleton C# példa

```
class WindowManager
{
    // Tárolja az egyetlen példányt, kezdetben null.
    private static WindowManager instance;

    // Glogális hozzáférést biztosító statikus művelet
    public static WindowManager GetInstance()
    {
        // Az egy példányt az első hozzáférés alkalmával hozzuk létre
        if (instance == null)
            instance = new WindowManager();
        return instance;
    }
    // Védett konstruktor
    protected WindowManager() { }

    // Az osztály egyik művelete
    public void DoSomething() { /*...*/}
}

…
// Valahol a kódban hozzáférünk hozzá a WindowManager singleton példányhoz
WindowManager.GetInstance().DoSomething();
```



Magyarázat

Singleton C# példa magyarázat

- Az egyetlen példányt maga az osztály tárolja egy statikus, pl. **`instance`** nevű védett (private) változóban.
- Az egyetlen példányt a többi osztály egy statikus, pl. **`GetInstance`** nevű metódussal éri el
  - Ez az első hívás alkalmával hozza létre a példányt és el is tárolja a statikus instance változóban
  - A későbbi hívások során ugyanezt a példányt adja vissza
- Az osztály konstruktora védett (private vagy protected), így más osztályok nem tudják a new operátorral példányosítani. Ez garantálja, hogy más osztály nem tudja megkerülni a **`GetInstance`** használatát, így további példányokat nem tud létrehozni.
- Mivel a **`GetInstance`** statikus, a kódban a **`WindowManager.GetInstance()`** hívással bárhol el tudjuk érni az egyetlen **`WindowManager`** objektumot.

Singleton

- C#-ban a hozzáférő statikus művelet helyett gyakori a statikus property használata (az alábbi példa csak a különbséget mutatja):

```
class WindowManager
{
    …
    // Glogális hozzáférést biztosító statikus művelet
    public static WindowManager Instance
    {
        get
        {
            // Az egy példányt az első hozzáférés alkalmával hozzuk létre
            if (instance == null)
                instance = new WindowManagerP();
            return instance;
        }
    }
    …
}

…
// Valahol a kódban hozzáférünk hozzá a WindowManager singleton példányhoz
WindowManager.Instance.DoSomething();
```

Szálbiztos C# implementáció *

- Nem kell tudni!
- A korábbi megoldásunk nem volt szálbiztos, vagyis több szálú környezetben kis valószínűséggel, de előfordulhat, hogy több objektum jön létre
- A statikus tagváltozók inicializálása az osztály első használatakor történik
- A C# fordító olyan kódot generál az inicializáláshoz, ehhez, ami szálbiztos.

```
class WindowManager
{
    // Tárolja az egyetlen példányt
    // Itt inicializáljuk, ez szálbiztos
    private static WindowManager instance
          = new WindowManager();

    // Glogális hozzáférést biztosító statikus property
    public static WindowManager Instance
    {
        get
        {
            return instance;
        }
    }
    // Védett konstruktor
    protected WindowManager() { }

    // Az osztály egyik művelete
    public void DoSomething() { /*...*/}
}
```

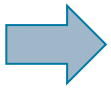
Singleton

- Használatát nem szabad túlzásba vinni
- Sok esetben használata ellenjavallt (anti-pattern), mert a **`GetInstance`** mindig az osztály típusával tér vissza, nem tudunk egy alternatív „dummy”/fake/mock implementációval visszatérni, ami pedig a unit tesztelésnél sokszor alapelvárás.
  - A Dependency Injection általában egy sokkal jobb tervezési minta!

Abstract Factory (Absztrakt gyár)

Abstract Factory

- Célja
  - Interfészt biztosít ahhoz, hogy egymással összefüggő objektumok családjait hozzuk létre anélkül, hogy specifikálnánk a konkrét osztályaikat
  - Így az objektumok létrehozása egy interfészen keresztül történik, a kódunk nem fog függeni a létrehozott objektumok konkrét osztályától/típusától

Próbáljuk megérteni egy példa segítségével 

Abstract Factory példa

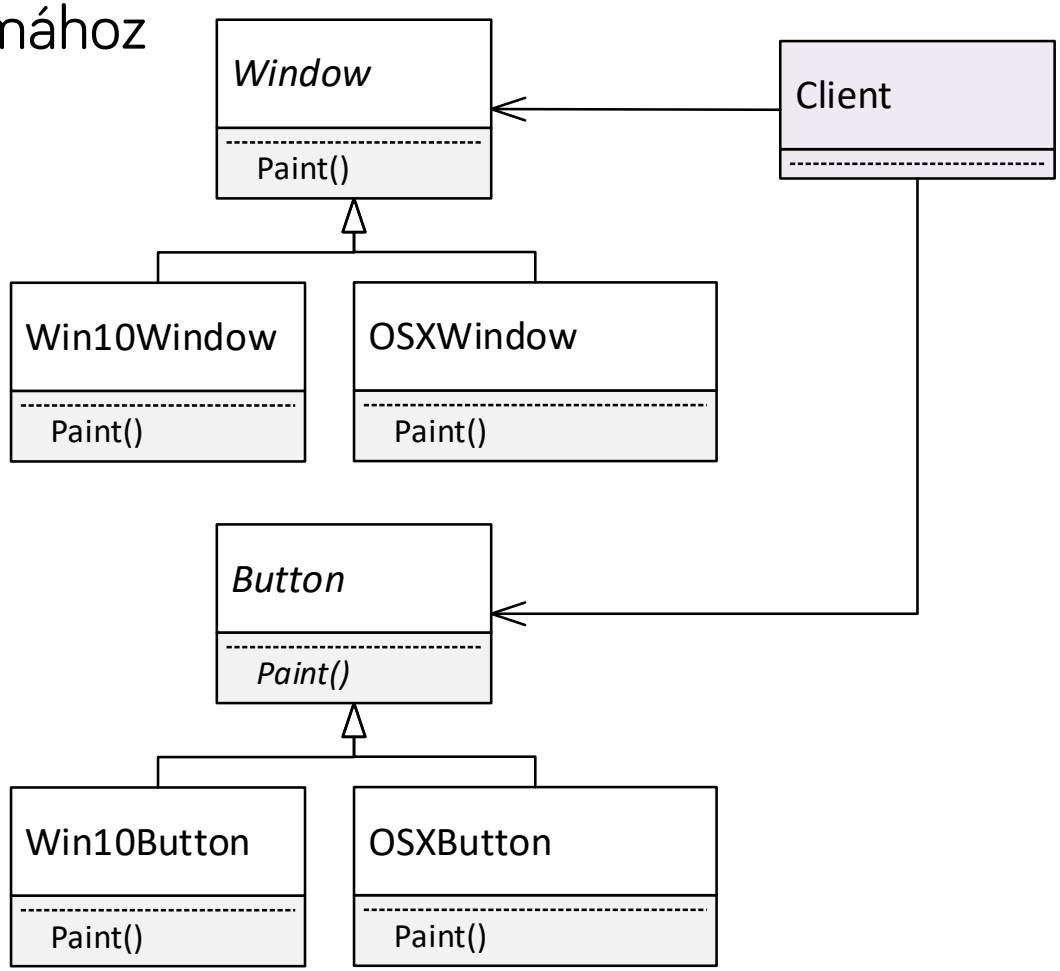
- A célunk
  - Egy modern ablakos/grafikus (GUI) felülettel rendelkező alkalmazás elkészítése
  - Az alkalmazás felülete felületelemekből komponálódik (pl. Window, Button, Checkbox, Scrollbar, …)
  - Az alkalmazásnak több megjelenési témát kell támogatnia („look-and-feel”, „skin”). Pl. Win10, WinClassic, OSX, stb.
  - A felületelemek megjelenése az aktuálisan kiválasztott megjelenési témát kell kövesse

Abstract Factory példa

- Első lépésben minden felületelemhez külön verziót készítünk minden megjelenési témához

A **kliens (Client)** reprezentálja az alkalmazás azon részét, mely a felületelemeket tárolja és példányosítja:

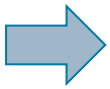
- Ez független kell legyen a konkrét megjelenítési téma osztályaitól (különben a téma nem lenne megváltoztatható).
- A Client emiatt interfészként vagy absztrakt ősként hivatkozik a felületelemekre



Abstract Factory példa

- Kód: lásd DesPattCode.AbstractFactory mappa. A kliens osztályt ilyen lépésekben fejlesztjük:
- **Client1_Initial** osztály: Ez a kiindulási verzió: ebbe még teljesen be vannak égetve a téma specifikus osztályok. Új témára átálláskor a kódját jelentősen módosítani kell.
- **Client2_UsingInterfaces** osztály: A kiindulási verziót úgy alakítottuk át, hogy a GUI elem hivatkozásokat közös ősre/interfészekre cseréltük. De a GUI elemek példányosítása még a new operátor használata miatt mindig téma specifikus, így új témára átálláskor a kódját jelentősen módosítani kell.
- **Client3_UsingAbstractFactory** osztály: A GUI elemek példányosítása már az Abstract Factory mintával történik, a példányosítás is téma független lett. Új témára átálláskor a kliens kódját így már nem kell módosítani.

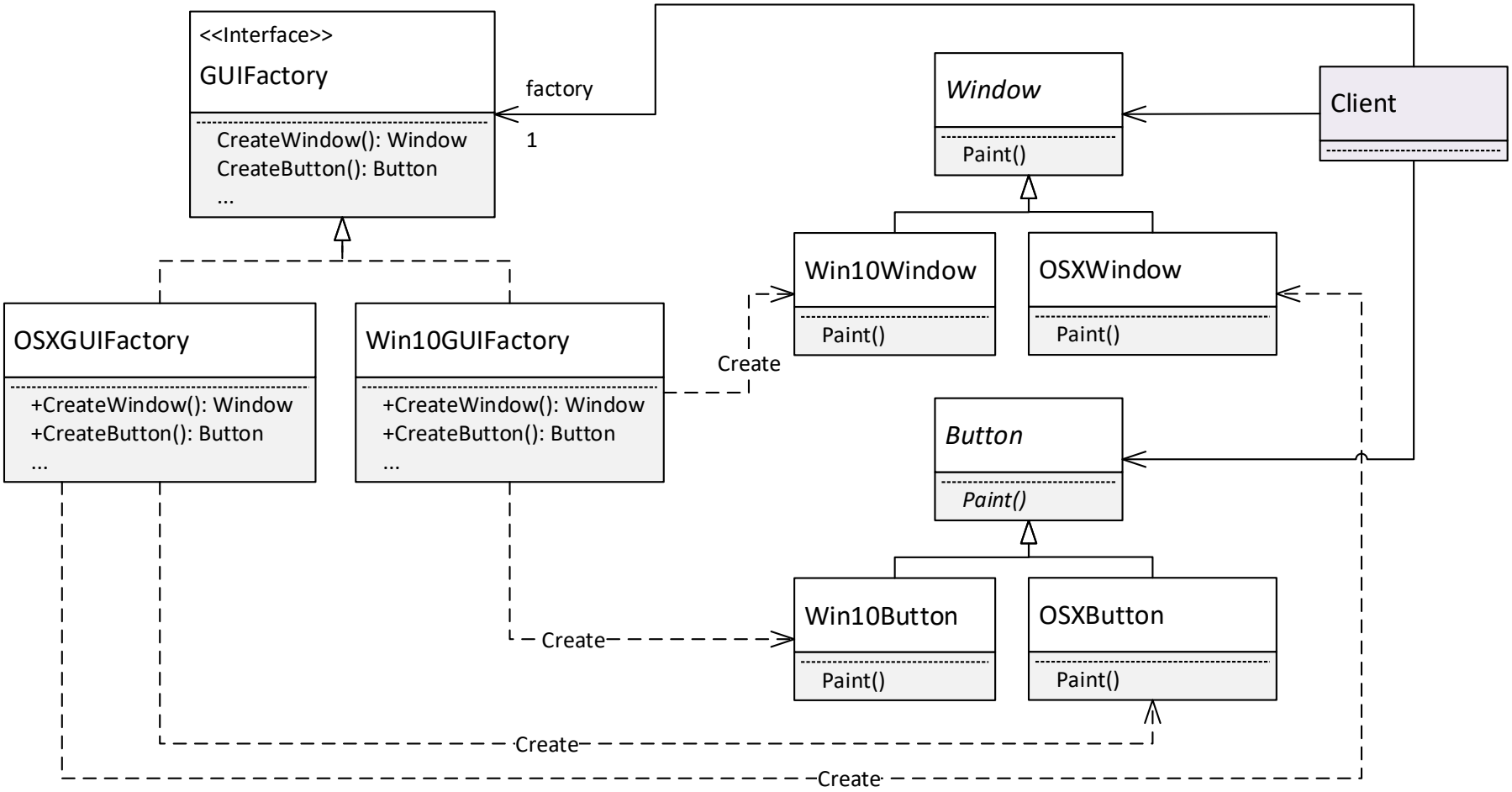
Abstract Factory osztálydiagram + magyarázat (+ mindenképpen érdemes Visual Studioban vagy GitHubon a teljes kódot megnézni!)



Abstract Factory - példa

- Ne drótozzuk bele a felhasználói felület elemeket az alkalmazásba. Sem az elemek **létrehozását**, sem pedig a használatukat.
  - Zárjuk egységbe ezek létrehozását, bízzuk más (abstract factory) objektum(ok)ra
  - Az alkalmazásban csak egy interfészen keresztül hivatkozzunk rájuk (így az implementációjuk kicserélhető)

Abstract Factory struktúra



Abstract factory - Megoldás

- Vezessünk be egy **GUIFactory** interfészt felületelemek létrehozására.
- A **GUIFactory** mindegyik művelete egy a művelet nevének megfelelő felületelemet gyárt és azzal tér vissza (általános, témafüggetlen típusként tér vissza): pl. **CreateWindow()** → **Window** visszatérési típus, **CreateButton()** → **Button** visszatérési típus).
- Mindegyik megjelenítési témához bevezetünk egy **GUIFactory** implementációt: (Win10 → **Win10GUIFactory**, OSX → **OSXGUIFactory**), amelynek metódusai az adott témához tartozó felhasználói felületelem objektumokkal térnek vissza. Pl. a **Win10GUIFactory.CreateWindow()** **Win10Window-t** ablakot példányosít, míg az **OSXGUIFactory.CreateWindow()** **OSXWindow**-t (a példányosítás a szokványos new operátorral történik).
- A klienst felkonfiguráljuk valamelyik **GUIFactory** implementációval (amelyik témát aktuálisan be kívánjuk állítani). A kliens eltárol erre egy hivatkozást egy tagváltozóban, a hivatkozás/tag típusa az általános **GUIFactory** interfész.

Abstract factory - Megoldás

- A kliens a felületelemek létrehozására az eltárolt **GUIFactory** objektumot használja a new operátor helyett: pl. ablak létrehozására a **CreateWindow-t**, gomb létrehozására a **CreateButton-t** hívja. Ezek a műveletek a beállított/eltárolt **GUIFactory** implementációnak (amire előzetesen felkonfiguráltuk) megfelelő témájú konkrét felületelem objektumokat hoznak létre.
- A kliens a létrehozott felületelem objektumokra csak az közös interfészükön/osztályukon keresztül hivatkozik.
- Összegezve: a kliens mind a létrehozott témafüggő felületelem osztályokra, mind a felületelemek létrehozására használt témafüggő **GUIFactory** implementációra csak a témafüggetlen közös interfészükön/ősosztályukon keresztül hivatkozik, a kliens kódja teljesen témafüggetlen lesz: a kliens kódjában semmilyen témafüggő kód/osztály nem szerepel. Így a téma megváltoztatásakor a kliens kódján nem kell változtatni semmit.
- Megjegyzés: a példában a felületelemeknek közös őse van, de lehetne közös interfésze is, feladatfüggő, melyik megközelítést célszerű változtatni

Abstract Factory kód

- GUIFactory

```
// Abstract Factory interfész.
// Mindegyik művelete egy a művelet nevének megfelelő vezérlőt gyárt.
// Mindegyik művelet visszatérési típusa az edott vezérlő
// interfész/közös ős típusa (vagyis független a témától!).
interface GUIFactory
{
    Window CreateWindow();
    Button CreateButton();
    Scrollbar CreateScrollBar();
}
```

Abstract Factory kód

- Win10GUIFactory és OSXGUIFactory

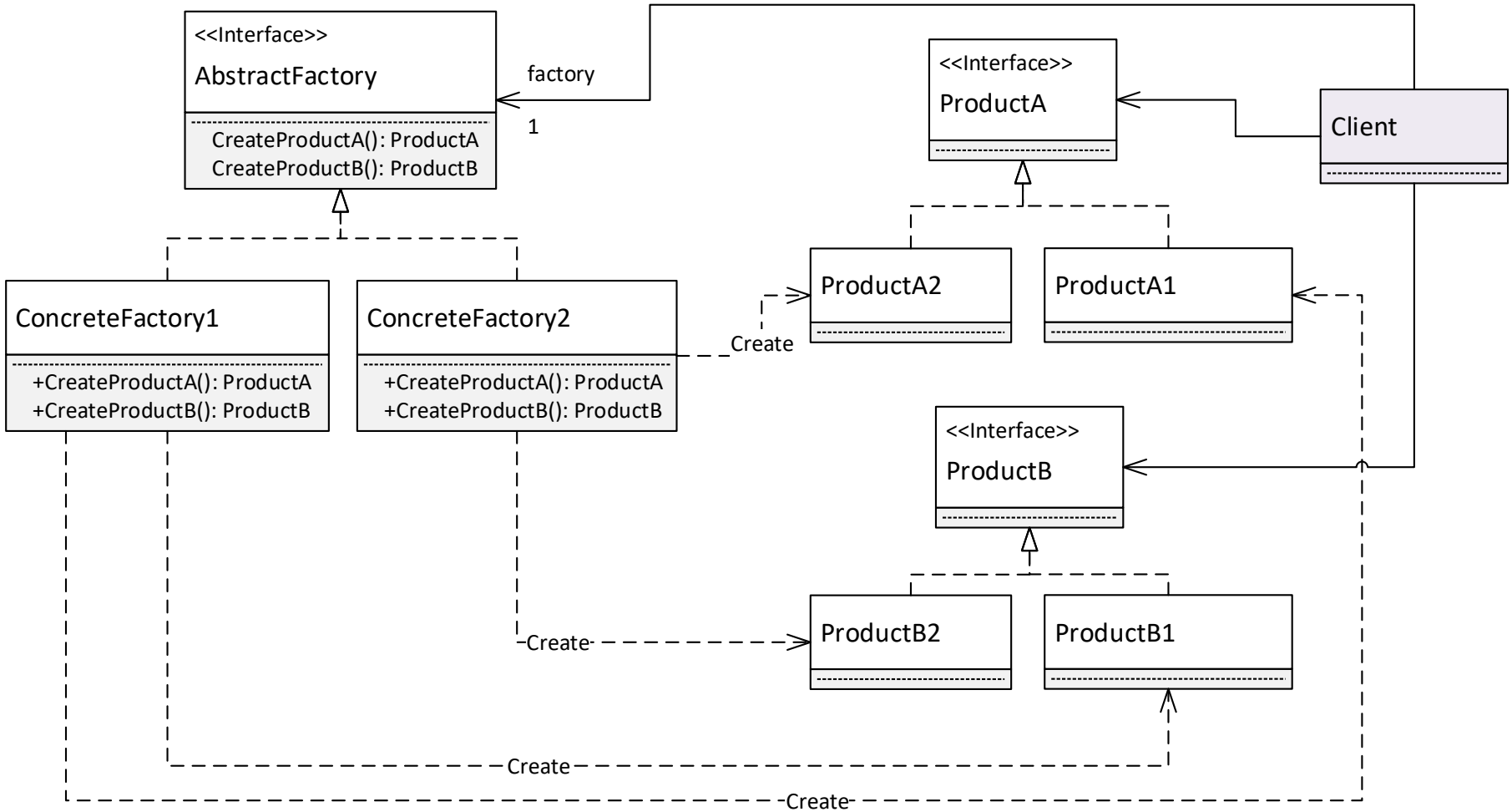
```
// A GUIFactory Win10 implementációja, Win10 témájú felületelemeket gyárt.
// Minden művelete egy a művelet nevének megfelelő típusú, az osztály
// nevének megfelelő (jelen esetben Win10) témájú felületelemet gyárt.
class Win10GUIFactory : GUIFactory
{
    public Window CreateWindow() { return new Win10Window(); }
    public Button CreateButton() { return new Win10Button(); }
    public Scrollbar CreateScrollBar() { return new Win10Scrollbar(); }
}

// A GUIFactory OSX implementációja, OSX témájú felületelemeket gyárt.
// Minden művelete egy a művelet nevének megfelelő típusú, az osztály
// nevének megfelelő (jelen esetben OSX) témájú felületelemet gyárt.
class OSXGUIFactory : GUIFactory
{
    public Window CreateWindow() { return new OSXWindow(); }
    public Button CreateButton() { return new OSXButton(); }
    public Scrollbar CreateScrollBar() { return new OSXScrollbar(); }
}
```

Abstract Factory kód

- Client3_UsingAbstractFactory

```
class Client3_UsingAbstractFactory
{
    // GUI elemek
    private Window wnd;
    private Button button;
    // ...
    // Factory interfész, ezt használja majd a kliens a GUI elemek létrehozásához.
    private GUIFactory factory;

    public void SetFactory(GUIFactory fy) {  // Művelet a factory beállításához
        factory = fy;
    }

    // Felületelemek példányosítása new helyett a factory segítségével
    public void InitGUIElements()  {
        wnd = factory.CreateWindow();
        button = factory.CreateButton();
        //...
    }

    public void DoSomethingComplex()  {
        // Demonstráljuk a GUI elemek kirajzolását
        wnd.Show(); wnd.Paint(); button.Paint();
        //...
    }
}
```

Abstract Factory – általános struktúra



Abstract factory

- 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)

Abstract factory

- 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 az előző példában minden termék egy közös GraphObject osztályból származik)

A létrehozási minták áttekintése

- A létrehozási minták célja: hozzuk létre az objektumokat úgy, hogy a rendszerünk rugalmasabb, könnyen bővíthető, a meglevő osztályok könnyebben újrafelhasználhatók legyenek
  - Abstract factory: külön factory hierarchia létezik termékcsaládok létrehozására
  - Singleton: csak egy példány létezzen, ami bárhonnan közvetlenül elérhető
  - Factory method (nem tanagyag): a leszármazottra bízza az objektum létrehozását, a Template Method minta alkalmazása objektum létrehozására.
  - ...

- A modern nyelvekben a reflexiós technikák segítségével a legtöbb létrehozási minta kiváltható, ma már gyakran ezt is használjuk
  - A létrehozandó típus reflexió esetén ugyanis sztring formájában is megadható, ami tetszőleges módon összeállítható
  - A reflexió hátrányai:
    - A többi megoldáshoz képest jelentősen lassabb. Ez van, amikor számít (nagy számú objektum létrehozásánál), van, amikor nem.
    - Nem típusbiztos, ha elgépelünk egy típusnevet, akkor csak futás közben derül ki, fordításkor nem.