Szoftvertechnológia és -technikák

9. Előadás

Architekturális tervezés

BME AUT
Automatizálási és Alkalmazott Informatikai Tanszék

Tartalom

- Tervezési szemlélet
- Alapvető architekturális minták
  - Rétegek
  - MVVM
  - MVC
  - Document View

Mi az architektúra?

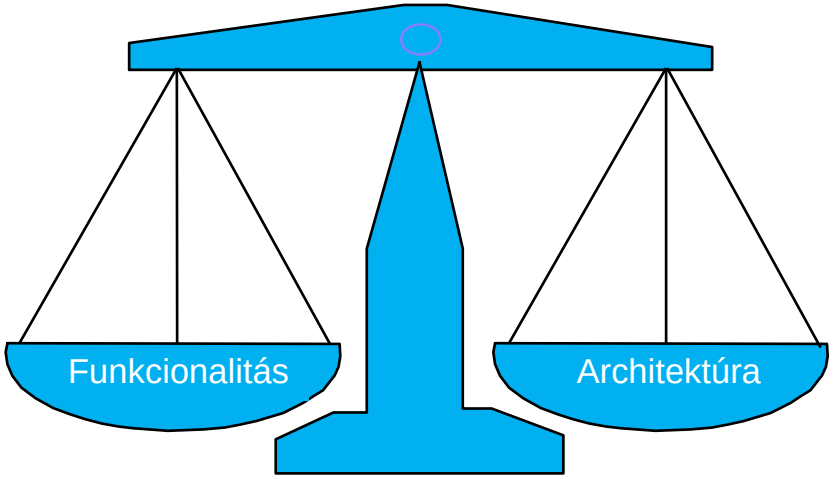
- A szoftverrendszer átfogó/magasszintű szervezése, strukturálása
  - A rendszert alapvető strukturális elemekre dekomponáljuk
  - Meghatározzuk ezen elemek kapcsolatát és együttműködését
- „From mud to structure.”
  - Az architekturális elem meghatározó a rendszer felépítése, teljesítménye, stb. szempontjából.
    - Amiből már nem lehet elvenni ahhoz, hogy megértsük és elmagyarázzuk a rendszer (alapvető) működését.
    - Nem foglalkozik az alacsonyszintű, belső kérdésekkel (modul belső felépítése, algoritmus, implementáció, stb.)

Architektúra tervezés

- Mikor?
  - A projekt kezdeti fázisában
  - A magasszintű követelmények meghatározásával párhuzamosan
- Nagy tapasztalati tudást igényel
  - Ha rosszul találjuk ki, később nagyon-(nagyon) nehéz és költséges architektúrát változtatni.

Az architektúra és a funkcionalitás

- Az architektúrának és a funkcionalitásnak egyensúlyban kell lenni!
  - Funkció: működjön jól! ←→ Architektúra: legyen jól strukturált!

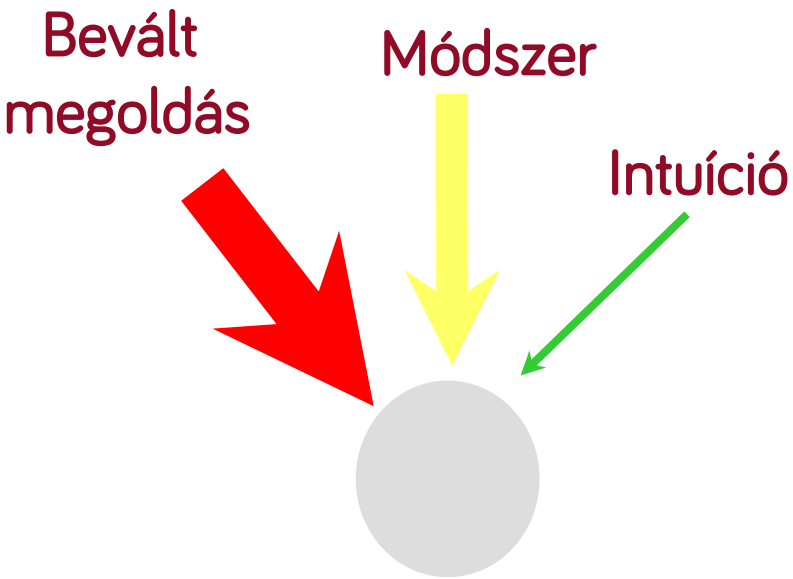


  - Az architektúra „léte” és jó megválasztása a rendszerek sikerének egyik alapkulcsa

Architektúra tervezés jelentősége

- Ha ad-hoc épül a rendszer (nincs architektúra):
  - A rendszer nagyon nehezen lesz:
    - Megérthető
    - Bővíthető
    - Karbantartható
    - Dokumentálható
  - Nem lesznek újrafelhasználható modulok/minták

Az architektúra forrásai



- Az esetek többségében választunk egy a projekt követelményekhez illeszkedő már bevált architekturális mintát
  - Vagyis nem „intuitíven” vagy „módszeresen” határozzuk meg az architektúrát.

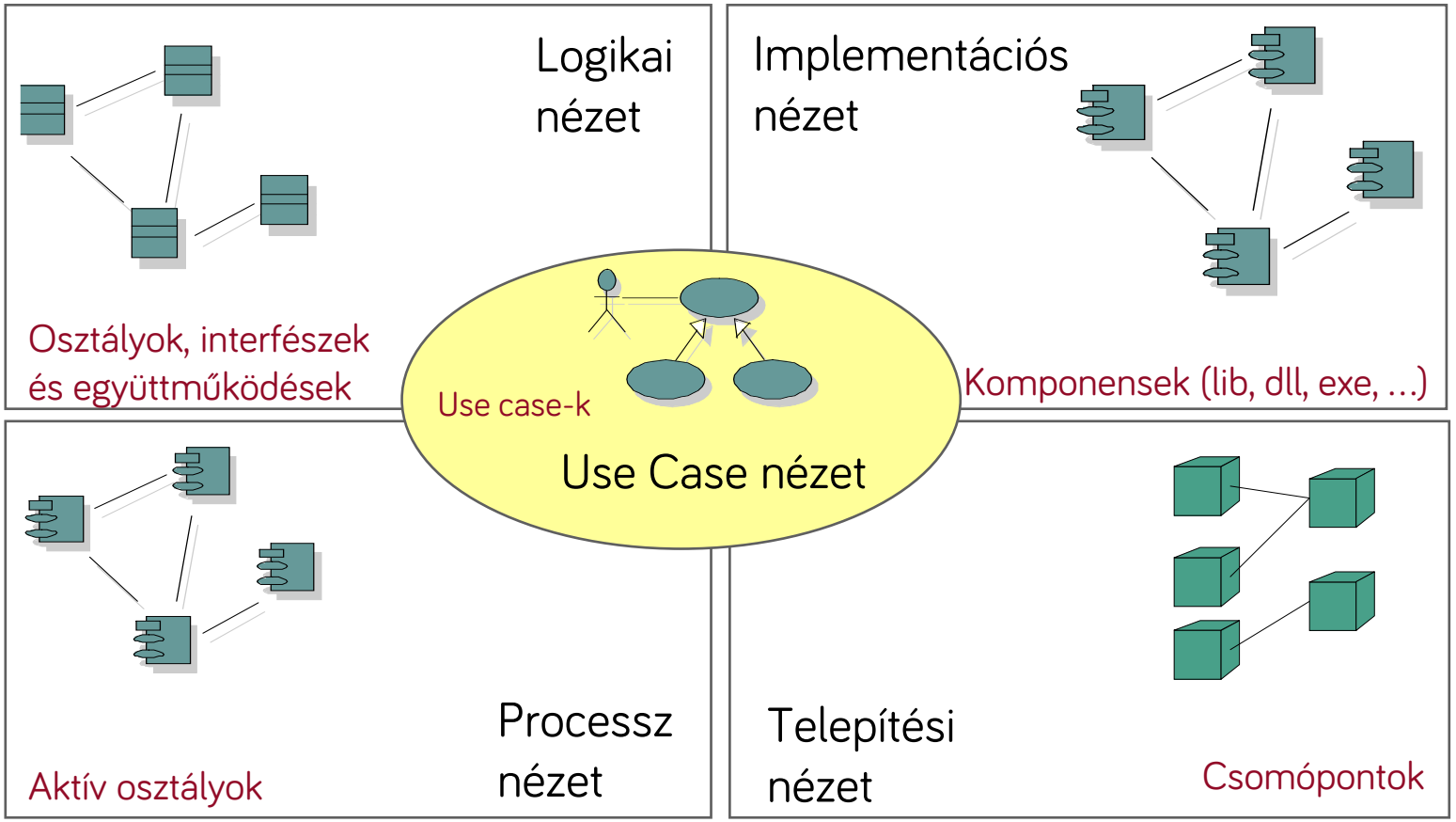
Pár architektúra minta példa

- Felhasználói felülethez kapcsolódók
  - Model-View-Controller (MVC)
  - Document-View
  - Model-View-ViewModel (MVVM)
- Általános
  - Pipes and filters
  - Layers
- Elosztott alkalmazások
  - Client-server
  - Three-tier architecture
  - Service-oriented architecture
  - Microservices architecture
- …

Jelölésrendszer

- Sokszor csak „dobozkákkal” és a köztük feltüntetett kapcsolatokkal
- Különböző UML diagramok

Az architektúra 4+1 nézete (haladó)



Magyarázat: következő dia...

Az architektúra 4+1 nézete (haladó)

- **Használati eset**
  - Funkcionális követelmények.
- **Tervezési vagy logikai**
  - Főbb csomagok (névterek), alrendszerek, osztályok.
- **Implementációs nézet**
  - Komponensek, dll-ek, exe, JAR csomagok, forráskódok szervezése.
- **Processz nézet**
  - Konkurens aspektus, folyamatok, szálak, holtpont, startup, shutdown, teljesítmény, skálázhatóság.
- **Telepítési nézet**
  - A futtatható és egyéb komponensek mely számítási csomópontokra (pl. számítógépekre, virtuális gépekre) kerülnek.

Alapelvek

UI és logika különválasztása

Separation of Concerns tervezési alapelv

Alapelv – UI és logika különválasztása

- **A felhasználó felület kódját (UI) és az alkalmazás/üzleti logika kódját válasszuk külön!**
  - Ne kódoljunk mindent bele az eseménykezelőkbe
  - alkalmazás/üzleti logika alatt legtöbbször az **alkalmazás adatainak** kezelését értjük

- A lényeg a függőség iránya: **a logika legyen független a felhasználói felülettől**:

- https://github.com/bzolka/AUT-SZTT repóban a ArchCode/SeparateUIAndLogic mappa

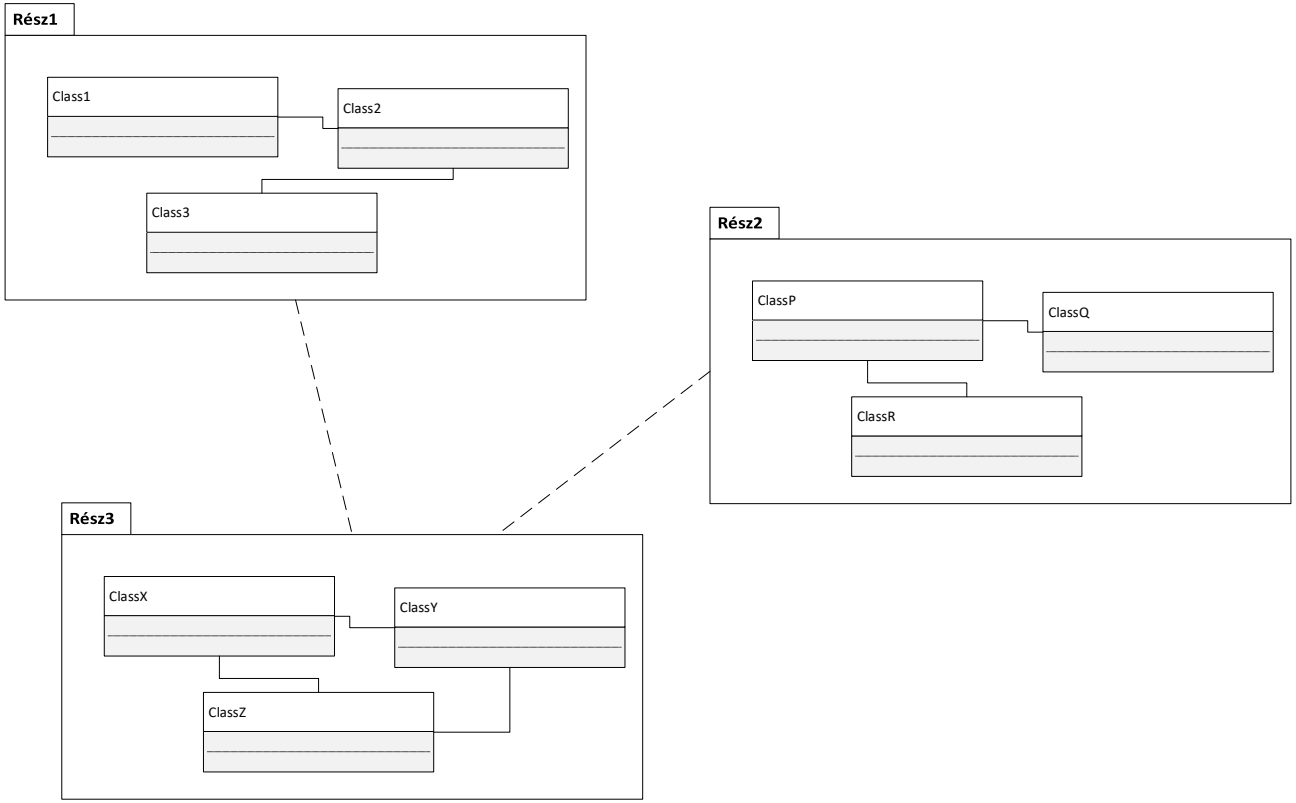
Demo

Alapelv - Separation of Concerns, SoC

- Vonatkozások/felelősségi körök különválasztása
- Elve: a kódrészeket (osztályokat) aszerint különítsük el, hogy milyen vonatkozású/felelősségi körbe tartozó feladatot végeznek
  - Az ugyanazzal foglalkozó kód egy körbe
  - A körök minél kevésbé lapolódjanak át és függjenek egymástól
- **A UI és logika (mint felelősségi körök) különválasztása egy példa erre**
- Példák felelősségi körökre
  - Technikai felelősségi körök: megjelenítés, felhasználói bemenet kezelése, üzleti logika, adatok kezelése, adatok tárolása (perzisztencia), külső rendszerekkel való kommunikáció, naplózás, stb.
  - Funkcionális: pl. egy webshop esetében vevőadatok kezelése, megrendelés kezelése, fizetés, számlázás, stb.

Separation of Concerns, SoC

- A SoC **laza csatoláshoz** (loose coupling) és **nagy kohézióhoz**/ösz-szetartáshoz (high cohesion) vezet.



- Egy felelősségi körön belül „high cohesion” – a kód/osztályok ugyanarra fókuszálnak
- A felelősségi körök/részek között „loose coupling” – a részeknek keveset kell tudni egymásról, kicsi a függőség

SoC tipikus példa

- SoC tipikus példa:
  - UI és üzleti/alkalmazás logika felelősségi kör különválasztása
  - A különválasztás után nagyobb a kohézió. Miért is?
    - Az egyes részek egy dologra fókuszálnak
    - UI kód NEM tartalmaz (üzleti/alkalmazás) logikát
    - A logika kód NEM tartalmaz felhasználó felülethez tartozó „logikát”

    Vagyis nem keverednek a különböző felelősségű részek.

SoC előnyök (UI és logika különválasztás példával illusztrálva)

- #1. Az egyes részek önmagukban könnyebben megérthetők, átláthatók (mivel egy dologra fókuszálnak)
- #2. Az egyes részek külön/párhuzamosan fejleszthetők (nagy projektben külön UI-hoz értő és „logikához” értő szakértő fejlesztők)
- #3. Könnyebb bővíteni, karbantartani (pl. a UI és a logika egymástól „függetlenül” bővíthető, lecserélhető)
  - Pl. pár év után lecserélődik az UI technológia, a logika nagy része változatlan maradhat (nem kell „hozzányúlni”, tesztelni)
- #4. Újrafelhasználhatóság (a UI és a logika egymástól „függetlenül” újrafelhasználható)
- #5. Az egyes részek önmagukban unit tesztelhetők (pl. az üzleti/alkalmazáslogika a UI-tól függetlenül)

Különválasztás módja

- Esetfüggő, hogyan célszerű
- Lehetőségek (akár kombináltan)
  - Java
    - Package-ek
    - JAR csomagok*
  - .NET
    - Névterek
    - Mappák
    - Szerelvények (dll, exe)*

* - nem kell tudni

SoC összefoglaló

- Ne feledjük: a SoC elv teljesen általános, a UI és logika szétválasztása csak egy példa!
  - A SoC az egyik legfontosabb **általános** tervezői elv
- Osztályok szintjén is, de architektúra szinten kiemelten fontos
- A „részeket” szokás moduloknak, komponenseknek is nevezni
- SoC és SRP hasonló: kéz a kézben együtt …

Rétegek (layers)

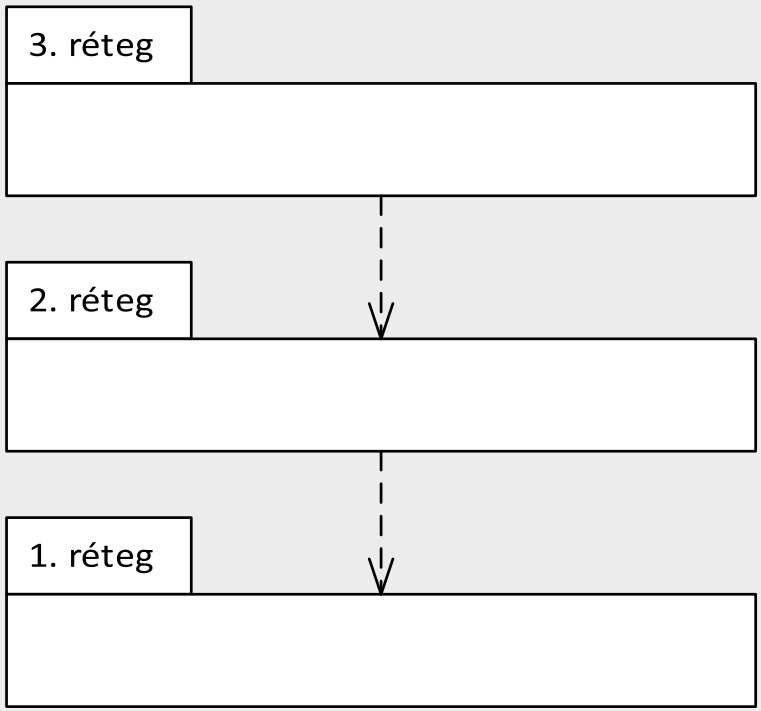
Rétegelés

- Legalapvetőbb szervezési elv
  - A kódot rétegekbe szervezzük
- Hol fordul elő
  - Hálózati protokollok
  - Példa: eszközmeghajtók -> operációs rendszer -> .NET/Java -> alkalmazás
  - Vállalati információs rendszerek (adatbázisban tárol adatokat, pl. a Neptun rendszer)
  - Stb.
- Ha a rendszer alacsony és magasabb szintű funkciók keveréke

Alapelvek

- Akárhány réteg lehet
- Egy adott réteg
  - szolgáltatásokat nyújt az a felette levő réteg számára
  - A saját szolgáltatásait az alatta levő réteg szolgáltatásaira építve valósítja meg
- Fontos a függőségek iránya: egy réteg függ az alatta levőtől, fordítva nincs függőség
- Jelölése: egyszerű „dobozkákkal”, vagy UML-ben csomagokkal

Példa három rétegre:



Előnyök – a SoC rétegekre vetítve

- Ha túl komplex a feladat: vezessünk be egy új absztrakciós szintet (új rétegbe) és oldjuk meg erre építve
- Egy réteg működése önmagában is megérthető, nem kell a többit is megértsük
- Külön/párhuzamos fejlesztés lehetősége
  - Az interfészek kialakítása után a rétegek egymástól függetlenül párhuzamosan fejleszthetők.
  - Az egyes rétegek eltérő technológiai ismeretekei igényelnek
    - (pl. UI fejlesztő, adatszakértő, stb.), nem kell mindenkinek mindenhez értenie a fejlesztőcsapatban.
- Könnyeb bővíthetőség, valamint az egyes rétegek újrafelhasználhatók. Pl. ugyanahhoz az üzleti logikai réteghez készíthetünk desktop és webes felületet (frontendet) is.
- Az egyes rétegek automata „unit” tesztelhetősége általában könnyebben megoldható.

Hátrányok

- Egyszerűbb feladatnál felesleges komplexitás
- A változások sok esetben kaszkádoltan végigvonulnak az összes rétegen, több helyen kell módosítani
  - Pl. felveszünk egy új adatbázis mezőt, akkor a UI és az adatbázis réteg között minden réteget módosítani kell
- Teljesítmény

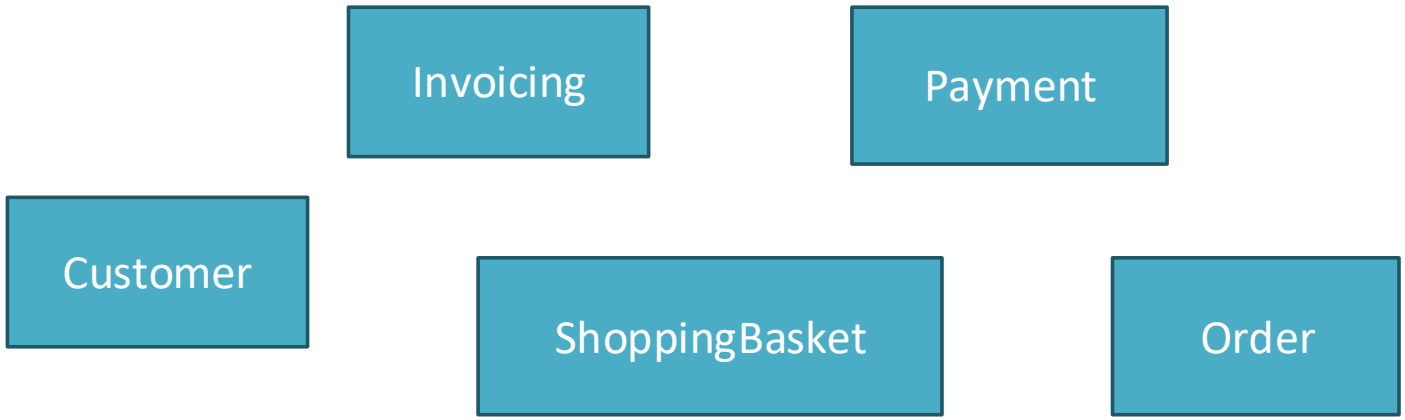
Többrétegű architektúra példák

- Kétrétegű
- Háromrétegű
- Kliens-szerver

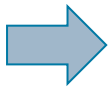
Kétrétegű architektúra

Egyrétegű - WebShop példa,

- Modulok
  - SoC, egyelőre csak funkcionális szétválasztás
  - Mindegyik valamilyen adatot tárol/kezel/jelenít meg

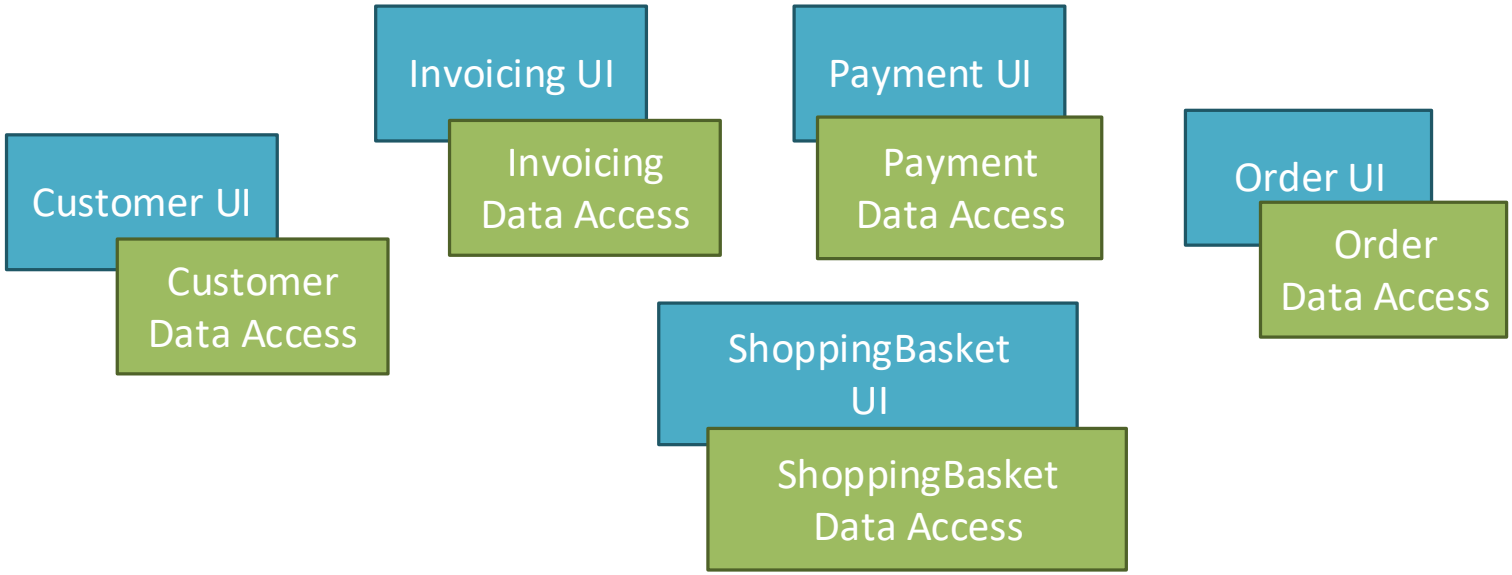


További szétválasztás



WebShop példa továbbfejlesztve

- SoC funkcionálisan
- Felhasználói felület (UI) és adathozzáférés (data access) vonatkozásában is különválasztva



Adathozzáférés: adatok tárolása, mentése, betöltése, vagyis perzisztencia

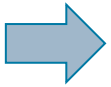
Adatthozzáférés

- Szerepe: adatok lekérdezése, mentése
- Data Access Layer, röviden DAL (sokan csak DAL-ként emlegetik).
- Perzisztencia rétegként is szokás rá hivatkozni
- .NET világában inkább „DAL”, Java világában inkább „Persistence” –ként hivatkoznak rá

Felhasználói felület szerepe

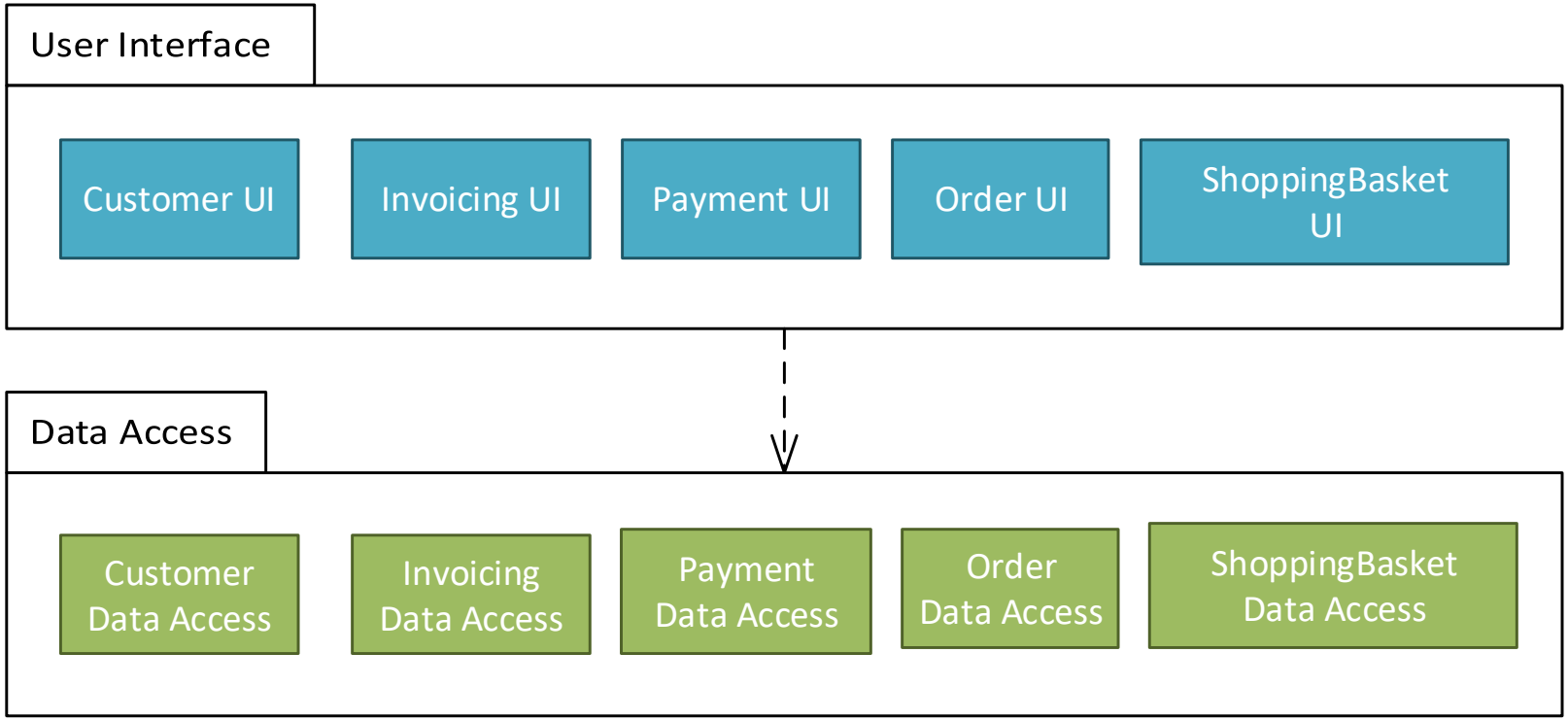
- Lehetővé teszi a felhasználó számára, hogy interakcióba lépjen az alkalmazással (pl. ablakok, vezérlők, weboldalak, konzol alkalmazásnál parancssor segítségével)
- Adatok megjelenítése a felhasználó számára
- A felhasználói kérések fogadása (lekérdezések és módosító parancsok).
- A felhasználó által megadott adatok (első körös) validálása

Szervezzük rétegekbe a korábbi alkalmazást



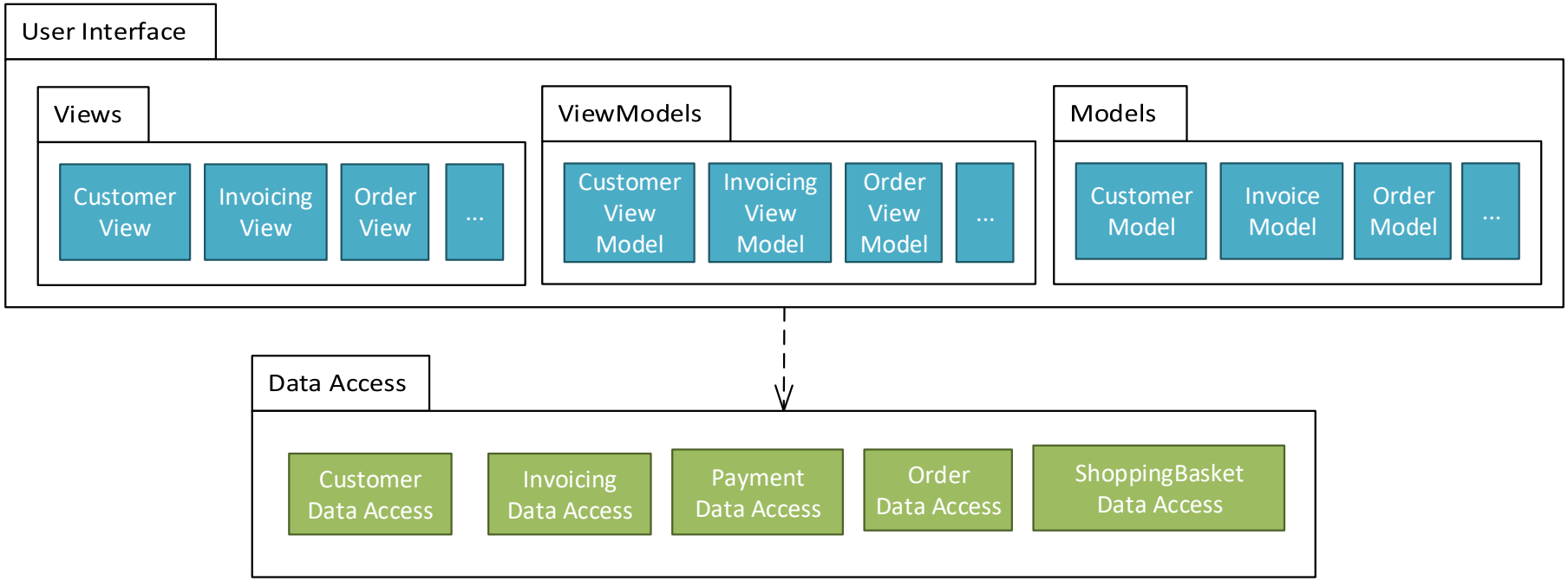
Kétrétegű WebShop

- Csoportosítsuk a UI és az adathozzáférési részeket két rétegbe



A rétegek belül tovább partícionálhatók

- Például UI rétegben MVVM minta (lásd később)



Kétrétegű előnyök

- Szokásos SoC előnyök, pár kiemelve
  - Ha le akarjuk cserélni a perzisztencia módját (pl. sima fájl, XML, SQL adatbázis, stb.), akkor csak a perzisztencia réteget kell lecserélni
  - Ugyanazokhoz az adatokhoz több különböző felhasználói felület készíthető

Milyen esetben használjuk a kétrétegű architektúrát?

- Olyan alkalmazásoknál, melyek adatokat kezelnek (jellemzően adatbázisban)
- Mikor elég a két réteg
  - Ha tipikusan egyszerű „minimál” adatműveletek vannak (Create/Read/Update/Delete – röviden CRUD)
  - Nincs komolyabb üzleti/feldolgozási/validációs logika az alkalmazásban
  - Gyorsan össze kell dobni egy egyszerű/prototípus alkalmazást
  - Egyszerűbb alkalmazások esetében használjuk bátran

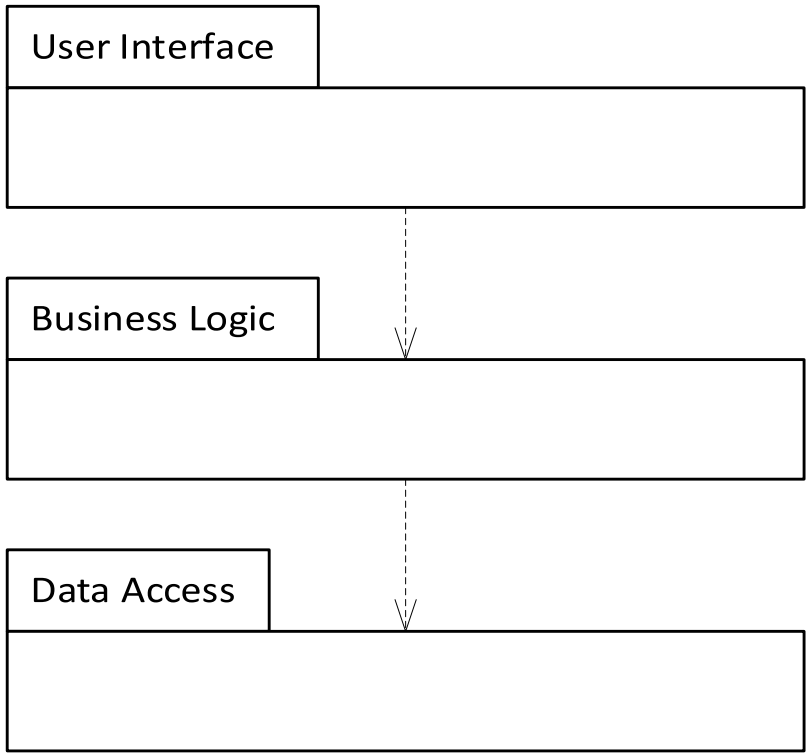
Milyen esetben használjuk?

- Mikor nem elég a két réteg
  - Ha az alkalmazásban jelentős az üzleti logika, akkor azt a SoC elvek alapján sem a UI, sem az adathozzáférési rétegbe nem praktikus tenni az üzleti logikát
    - Más a felelősségi kör
    - Pl. üzleti logika nem lesz megérthető, tesztelhető, bővíthető, újrafelhasználható a UI vagy DAL nélkül (attól függően, melyikbe tesszük), ha a kódban össze van nőve ezekkel.
    - A kétrétegű esetben gyakorlatban sokszor a UI-ba kerül, így viszont nem lehet ugyanahhoz a logikához különböző frontendeket (pl. web, desktop, mobil) írni.

Háromrétegű architektúra

Háromrétegű architektúra alapok

- Kibővíti a kétrétegű architektúrát egy üzleti logikai réteggel
- Business Logic Layer, rövidítve BLL

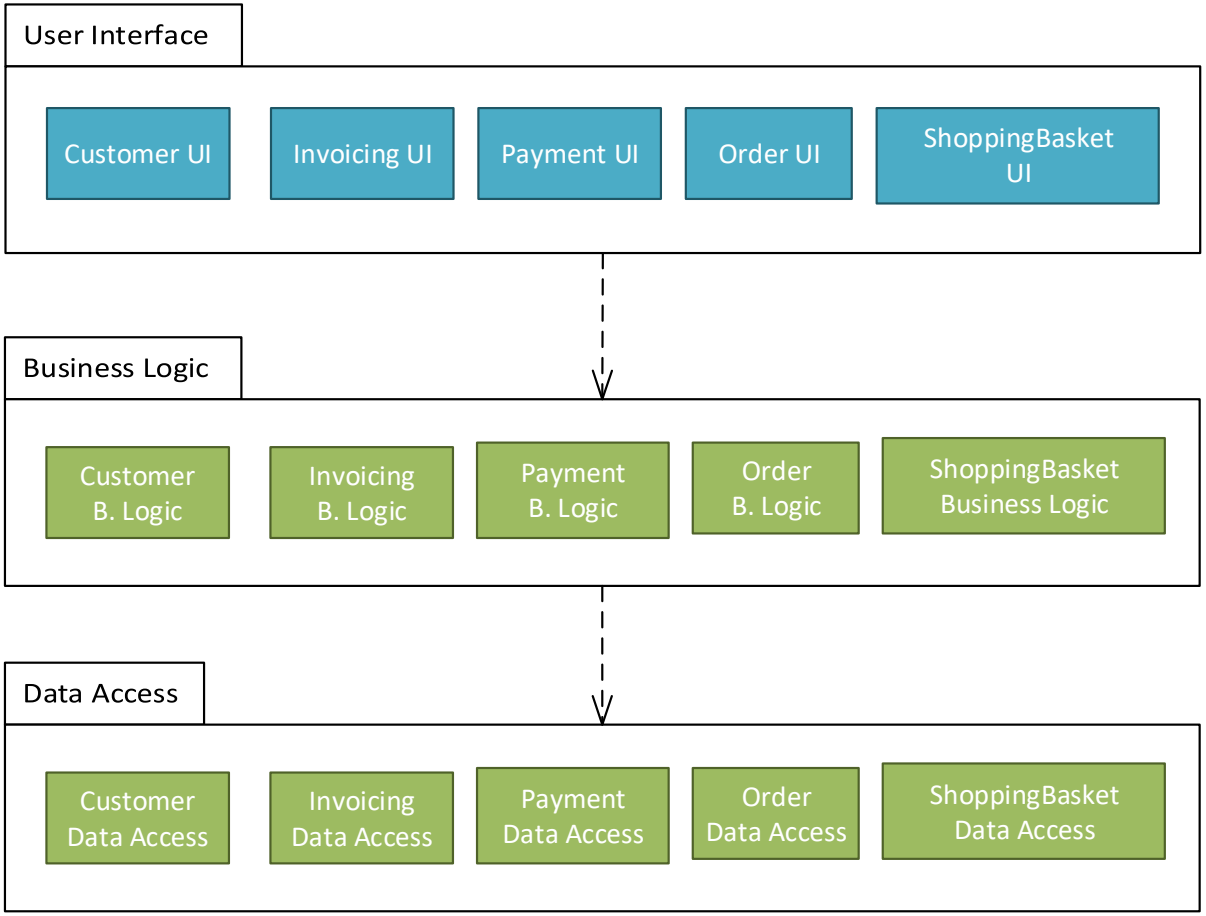


Üzleti logikai réteg

- Üzleti logikai szabályokat, validációs szabályokat, stb.-t tartalmaz valamint folyamatokat ír le
- Pl.:
  - **Egyszerű validációs szabály**: árucikk neve nem maradhat üresen; a születési dátum nem lehet a jövőben
  - **Logikai szabály**: pl. számla esetén kell legalább egy tétel; a végösszeg meg kell egyezzen a tételek összegével; bruttó ár számítása nettó árból; kedvezmény az összeg/vevő függvényében; két hallgató Neptun kódja nem lehet azonos
  - **Folyamatok**: egy online megrendelést előbb-utóbb ki kell szállítani és ki kell számlázni (vagy „cancel”-elni kell, ha nincs készleten áru).
- Független a megjelenítéstől

Háromrétegű WebShop

Egyszerűsítéssel



Az ábra egyszerűsít: a rétegeken belül gyakran nem ilyen egyszerű egymás mellé rendeltség van. Pl. a bruttó-nettó számítási szabályok közösek az Order és az Invoicing BLL modulokra, így ezt érdemes külön kódba kiszervezni, melyet az Order és az Invoicing is felhasznál.

Előnyök és hátrányok

- Előnyök
  - Szokásos SoC előnyök az üzleti logika vonatkozásában:
  - Az üzleti logika önmagában megérthető, fejleszthető, bővíthető, újrafelhasználható, tesztelhető
  - Újrafelhasználhatóság: sokszor egy alkalmazás/rendszer több modulja is felhasználja ugyanazt a logikát:
    - pl. nettó-bruttó számítás szabályait az Order és az Invoicing modulok is
- Hátrányok
  - Megnövekedett komplexitás, több munka (extra réteg)

Kódszervezés

- Esetfüggő, hogyan célszerű
- Lehetőségek (akár kombináltan)
  - Java
    - Package-ek
    - JAR csomagok*
  - .NET
    - Névterek
    - Mappák
    - Szerelvények (dll, exe)*

* - nem kell tudni

Felhasználói felület kialakításához kapcsolódó architektúrák

BME AUT

Felhasználói felület kialakításához kapcsolódó architektúrák

- Kétrétegű (felület és logika két külön rétegben)
  - Már láttuk az előző „fejezetben”
- MVC – Model-View-Controller
- MVVM – Model-View-ViewModel
- Document-View
- …

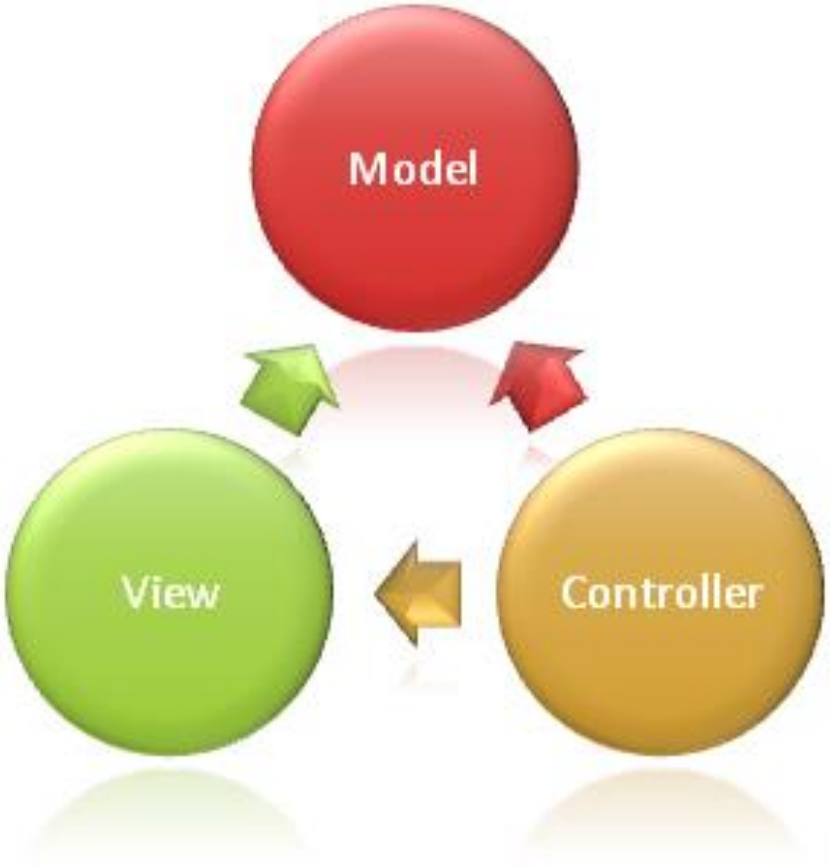
A Model-View-Controller (MVC) architektúra

Szereplők és függőségek

A kódot (osztályokat) három felelősségi körbe csoportosítjuk:

- Controller
- Model
- View

Függőségek:



Szereplők és függőségek

- **Controller**: fogadja a felhasználói bemenetet és interakciókat
  - Pl. egy webalkalmazásnál ez kapja meg a felhasználói interakciók hatására bekövetkező hálózati kéréseket a felhasználói által megadott adatokkal.
  - Fogadja a felhasználói adatokat és továbbítja a model számára.
  - A kiválasztja ki a megfelelő View-t (vagy View-kat) a megjelenítéshez
  - Átadja a Model megfelelő adatait a View-nak a megjelenítéshez

Szereplők és függőségek

- **Model**: az adatok reprezentálásáért, kezelésért felel (+ üzleti logika, validációs szabályok, stb.)
  - A felhasználói bemenetet a Controllertől kapja meg.
  - *Független a felhasználói felülettől!*
- **View**: az adatok megjelenítésért felel

Előnyök

- UI – logika (SoC) különválasztás előnyök mindegyike itt is él: lásd korábban! Többek között:
  - Nem keverednek a különböző felelősségű részek a kódban, könnyebb megérteni, karban tartani, tesztelni, …
  - Lásd függőségek ábra: a Model (vagyis a logika) nem függ sem a Controller-től sem a View-tól!

    A UI kód jellemzően gyakrabban változik, mint az alkalmazáslogika: ha nem lenne különválasztás, akkor minden UI változáskor a logika osztályait is újra kellene tesztelni.

Hátrányok

- Extra komplexitást jelent
  - Új absztrakciók megértése és az ehhez való illeszkedés
  - Kódban navigáció körülményesebb („ugrálás” az adott funkció model/view/controller részei között)
- Csak indokolt esetben használjuk
  - Ha jól illeszkedik az adott feladathoz !!!
  - Ha az általunk választott keretrendszer is támogatja.

Jellemzők

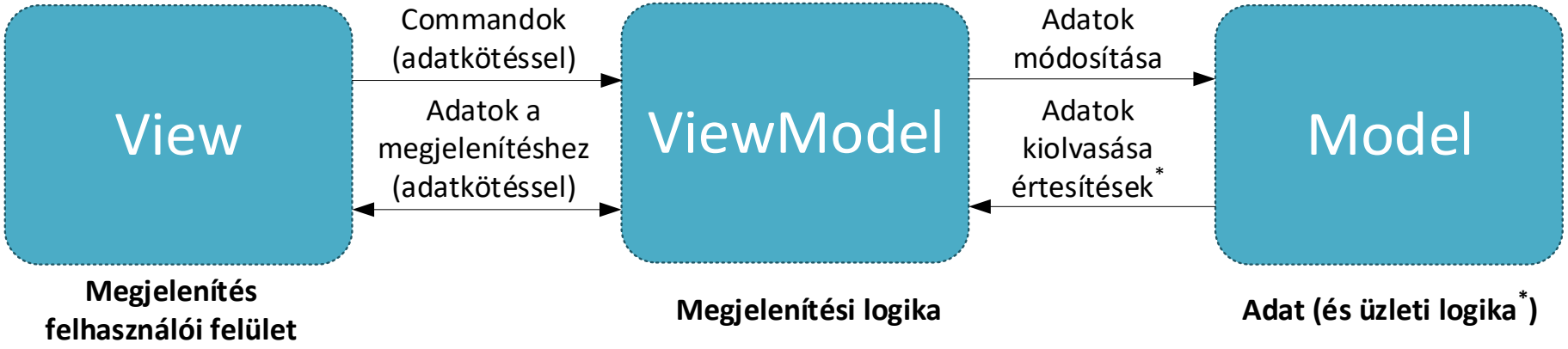
- A konkrét megvalósítása nagyon függ attól, hogy dektop/mobil/web környezetben alkalmazzák
- Ma a (tradicionális) webalkalmazások esetében gyakori:
  - Pl:
    - Java → Spring MVC
    - .NET → ASP.NET Core MVC
  - Desktop alkalmazásoknál már kevésbé elterjedt.
- *A gyakorlatban akkor használjuk, ha a keretrendszer, amit használunk, beépítve támogatja*

Model-View-ViewModel (MVVM) architektúra

Bevezetés

- Eseményvezérelt és vizuális programozás tárgyból
  - Részletesen
  - Gyakorlati példák
- Komolyabb platformszolgáltatásokat igényel
  - Pl. adatkötés (data binding)

Felelősségi körök



Felelősségi körök

- **Model**: **domain adatosztályok, adatokat reprezentálják**
  - pl. Person, Employee, Student, Order, OrderItem, Tweet, Contact
    - Ezek tagváltozókban/propertykben a szükséges információt hordozzák. Pl. Contact esetén Name, Address, PhoneNumber.
  - **Bizonyos esetekben az üzleti logika is ebben van** (pl. egy számológép app), más esetekben külön „service” osztályokban, melyeket a ViewModel használ, és a „service” osztályok kérdezik le és módosítják a Model adatosztályokat.
  - **Független a megjelenítéstől!**

Felelősségi körök

- **View**: csak megjelenítés (+ felhasználói események innen erednek)
  - Felületelemekből épül fel (pl. szövegdoboz, jelölőnégyzet, stb.),
  - Jellemzően valamilyen deklaratív leíró nyelvvel definiálható (pl. XAML).
- **ViewModel**: *megjelenítési* logika, összeköti a modellt és a nézetet
  - Adatkötés segítségével biztosítja a nézet számára a megjelenítendő adatokat (ehhez felhasználja a modell osztályait).
  - Command objektumok segítségével fogadja az interakciókat a nézet felől (ezek hatására transzformációkat végezhet és továbbítja a változásokat a modell felé). Pl. adott gomb lenyomása futtatja a View-ban a gombhoz kötött ViewModelben definiált parancsot.

Előnyök

- Lásd SoC előnyök
  - Kiemelve: a model osztályok függetlenek a megjelenítéstől!
- Párhuzamos fejlesztés lehetősége, szakértői tudás különválasztása
  - A „designer” megtervezi, kialakítja a View-t
  - A fejlesztők lekódolják a Modelt és ViewModelt
- A UI logika a ViewModelben van, leválasztva a megjelenítéstől (View), a ViewModel kódja unit tesztelhető

DOCUMENT-VIEW ARCHITEKTÚRA

BME AUT

Bevezető

- Az alkalmazások többsége adatokat jelenít meg, melyet a felhasználó módosíthat.
- Alapigazság: ne keverjük bele a UI-ba az alkalmazáslogikát (lásd korábban)
  - Válasszuk külön az adatok **kezeléséért** felelős kódot az adatok **megjelenítéséért** felelős kódtól.
- Egy lehetséges megoldás a Document-view architektúra
  - Alternatíva pl. a Model-View-Controler (MVC)

Bevezető

- Az MVC napjainkban a **webalkalmazások** esetében gyakori
- A Document-View a **desktop** alkalmazások estében használatos (dokumentum alapú alkalmazások esetén, pl. Word-höz hasonló)

Kapcsolat az MVC-vel (némi egyszerűsítéssel élve):

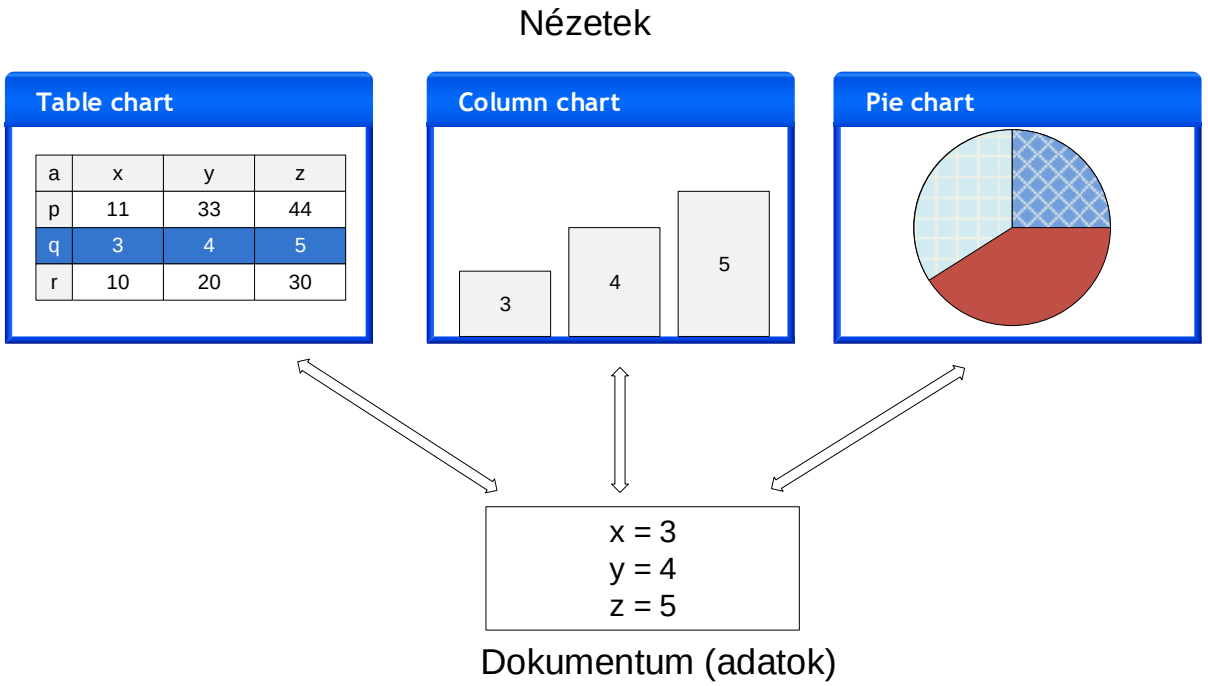
- Model → Document
- View+Controller összevonva → View

A Document-view architektúra szereplői

- Document (dokumentum)
  - Feladata az adatok tárolása, menedzselése.
  - Olyan osztály(ok), melyek az adatokat tagváltozóikban tárolják, és olyan tagfüggvényekkel rendelkeznek, melyek kezelik ezeket az adatokat (pl. Load, Save), és elérhetővé teszik más osztályok számára (pl. a View részére)
- View (nézet)
  - Feladata az adatok megjelenítése a dokumentum adatai alapján és a felhasználói interakciók kezelése (pl. menük, egér, billentyűzet).
  - A felhasználói interakciók során általában a nézet a dokumentum tartalmát módosítja
  - A view általában egy ablakként, tabfülként vagy egy „panelként” jelenik meg a kliensalkalmazásokban

Támogatja a következőket

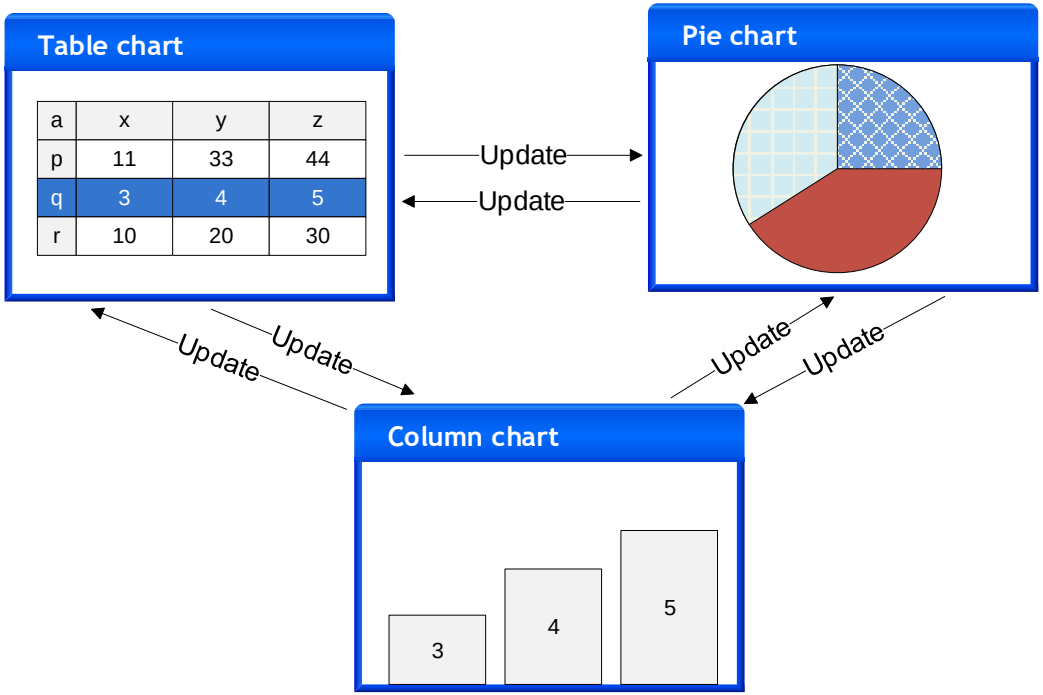
- Több dokumentum egyidejű megnyitása
  - pl. Firefox tabok, MS Word dokumentumok
- **Egy dokumentumhoz több és többféle nézet kapcsolódhat**
  - pl. Excel, de több alkalmazásban a View/New Window menüvel



Egy dokumentumnak több nézete lehet!

- Meg kell oldani, hogy az egyes view-k konzisztens nézetét jelenítsék meg az adatoknak.
- Például ha a felhasználó megváltoztatja az egyik nézeten az adatokat, frissíteni kell a többit. Hogyan? Közvetlen egymásra hivatkozás+függvényhívással (minden nézet tartalmazzon egy referenciát az összes többire, hogy változás esetén tudjon mindenki mást értesíteni)?

Demo (Word, VS)

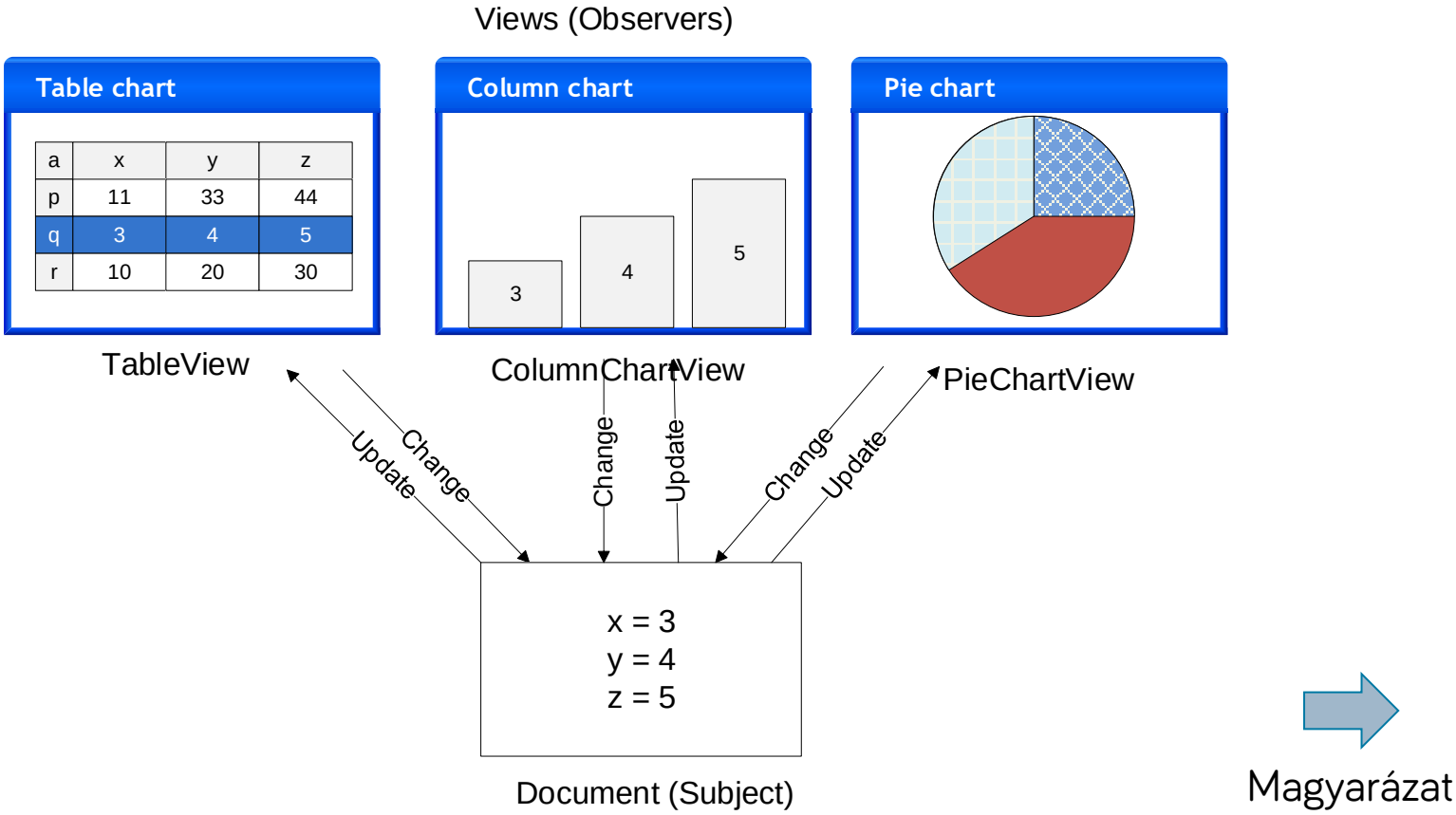


Nézetek frissítése

- Közvetlen hivatkozás+függvényhívás hátrányok
  - Függőség a konkrét osztálytól.
    - Pl. a TableView függ a ColumnChartView és a PieChartView osztályoktól
  - **Ha új nézetet szeretnék bevezetni, minden MEGLÉVŐ nézet osztályt módosítani kell ☹ ☹ ☹ ☹ ☹**
  - Az alkamazáslogika nem újrafelhasználható, mert össze van vonva a megjelenítéssel.
    - Cél lenne, hogy úgy jelenjen meg, hogy ne legyen benne hivatkozás egy (konkrét) megjelenítési osztályra sem, mert akkor fel tudnánk több helyen használni
  - Nehéz karbantartani, továbbfejleszteni, újrafelhasználni, mert túl szoros a csatolás az osztályok között

A nézetek frissítése

- A jó megoldás: Observer minta alkalmazása

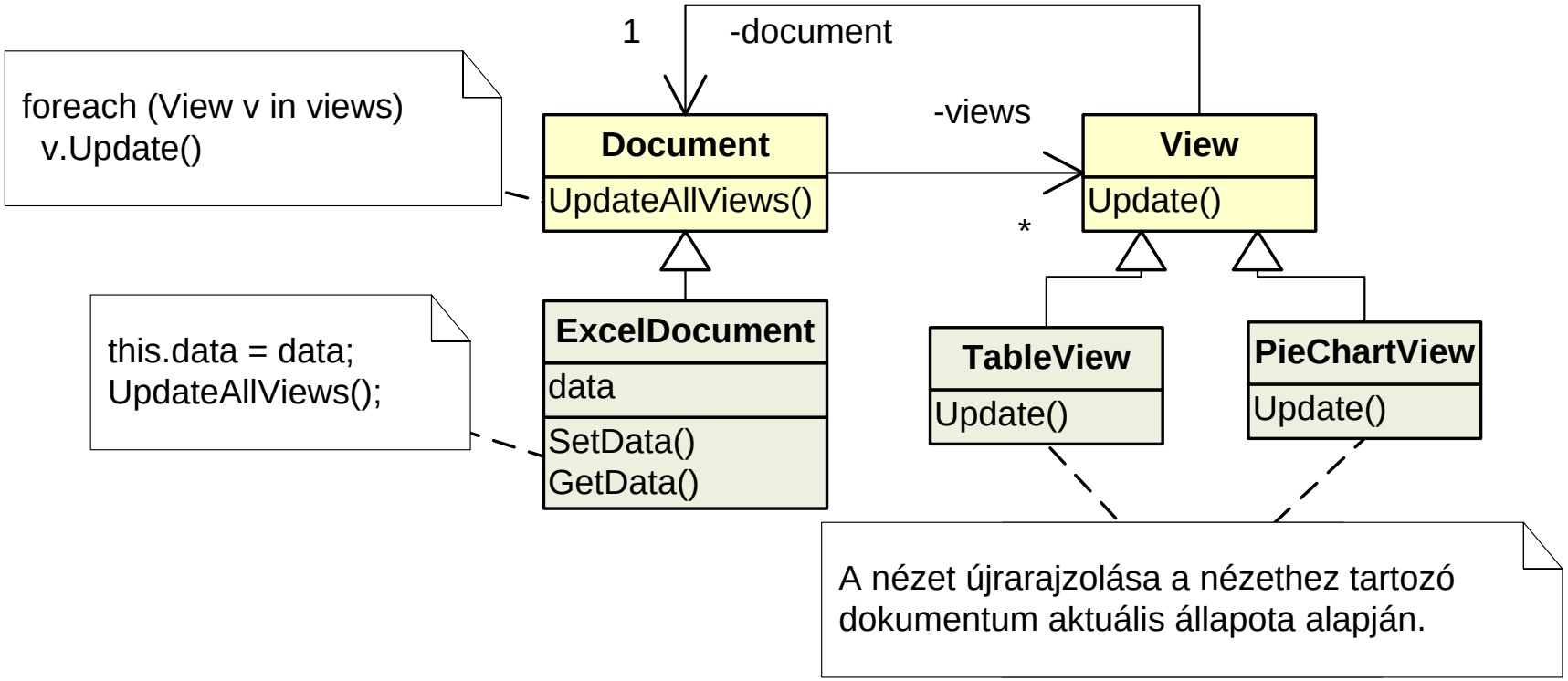


A nézetek frissítése

- Magyarázat
  - 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 lekérdezi a dokumentum állapotát és frissíti magát
  - A dokumentum csak egy közös View interfészen/ősosztályon keresztül tárolja a beregisztrált view-kat (nem függ az egyes típusoktól).

Statikus nézet (osztálydiagram)

- Példa (Excel)



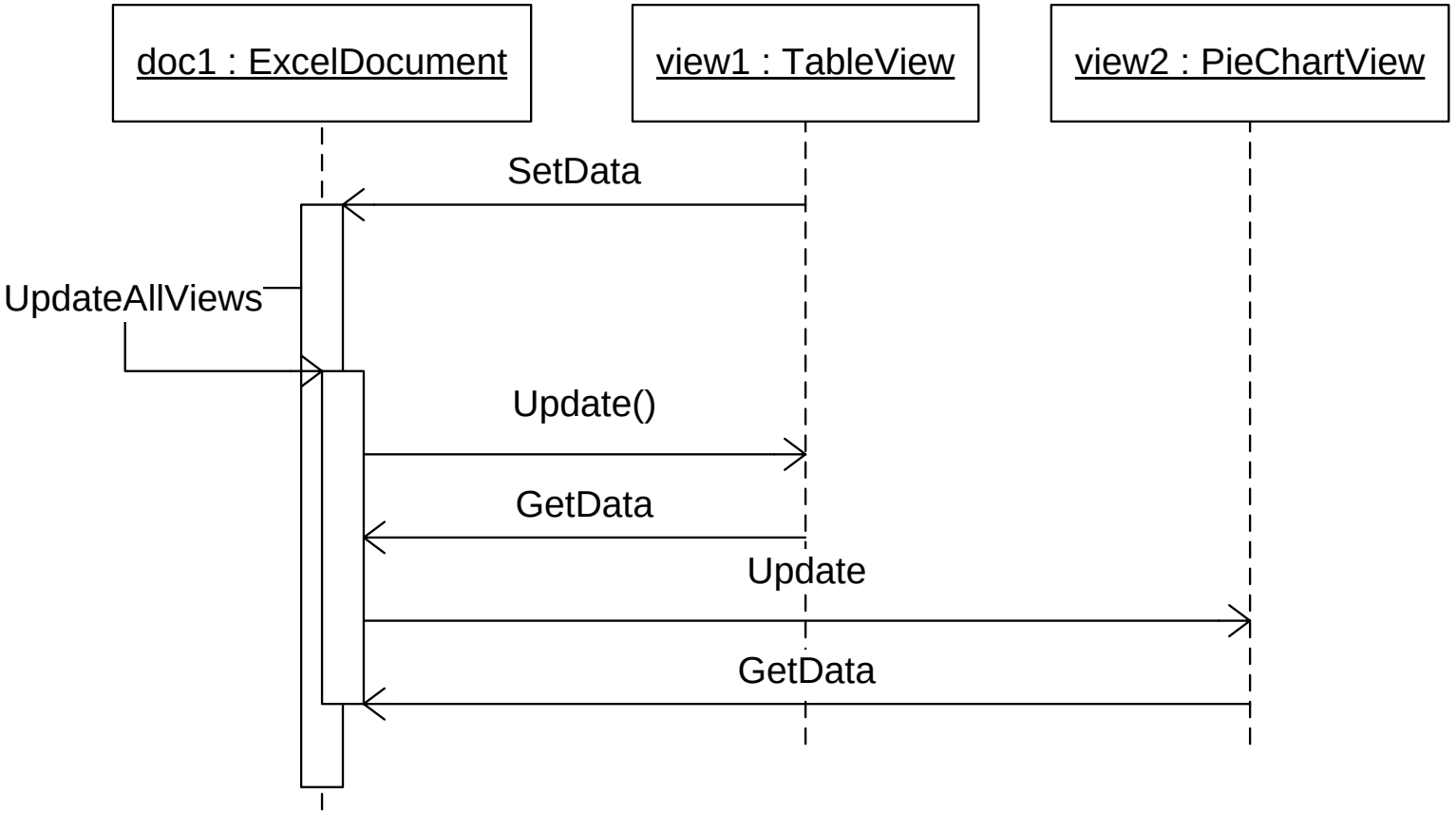
Osztálydiagram magyarázat

- 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.
  - Tartalmaz egy referenciát a dokumentumra, amin keresztül a leszármazott nézetek elérhetik a dokumentumot, melynek adatait megjelenítik.
- TableView, PieChartView, stb.
  - A dokumentum egyes nézeteit reprezentálják.
  - A View-ból származnak.
  - Felüldefiniálják vagy implementálják a View Update műveletét. Az Update műveletben a nézet a dokumentum aktuális állapota alapján frissíti magát.

Dinamikus nézet (szekvenciadiagram)



Konklúzió

- Document-view előnyök
  - Korábban megadot **SoC előnyök**
    - Pl. a document kódjában csak egy View lista van, így a document független az egyes View-t implementáló osztályoktól (laza csatolás!)
  - Egy egyszerű mechanizmust kaptunk arra, hogy az összes view **konzisztens nézetét** mutassa az adatoknak.
  - A rendszer **könnyen kiterjeszthető új view osztályokkal**. Sem a document, sem a többi view osztályt nem kell ehhez módosítani.
- Document-view hátrányok
  - Megnövekedett komplexitás
  - Csak indokolt esetben, ha illeszkedik a konkrét feladathoz