Rétegzett alkalmazás-fejlesztés

Klasszikus háromrétegű architektúrák

- Adat, logika és megjelenítési rétegből áll.
- A háromrétegű architektúrában a felső rétegek mindig a közbenső rétegeken keresztül érik el az alsó rétegeket.

Megjelenítési réteg

- Az alkalmazás legfelső rétege.
- Fő funkciója a rétegnek, hogy lefordítsa a feladatokat az alsóbb rétegek felé és hogy a visszajövő adatokat megjelenítse a felhasználó felé.

Logikai réteg

- Alkalmazás viselkedése található meg ebben a rétegben.
- Feldolgozza a felsőbb rétegtől jövő feladatokat és döntéseket hoz az üzleti logika alapján.
- Feladata még az adatmozgatás a két réteg között.

Adat réteg

- Tárolja az információkat.
- Feladata az adatbázissal és a fájl rendszerrel való kommunikáció megvalósítása és a megfelelő adatok kigyűjtése.
- Az információt amit kinyer, azt vissza is küldi a logikai rétegnek.

Fontos kritériumok

- **Legyen**
    - Karbantartható (maintainability)
        - Minimalizálni tudja a változtatáskor, javításkor és új funkciók bevezetésekor igényelt erőforrásokat.
    - Újrafelhasználható (reusability)
        - Pl.: DLL-t készítünk, amit feltudunk használni másik projekthez.
    - Skálázható (scalability)
        - Növekvő igények kiszolgálásának problémamentes kezelését jelenti.

Konverterek

- Külön adatkonvertálási rétegként lehet használni, ami a **ViewModel** és a **View** között helyezkedik el.
- Szükség akkor lehet rá, ha valamilyen speciálist formázást kell végrehajtani az adatokon, amiket a **ViewModel** nem biztosít.

Felhasználói felület alkalmazása a Model-View-ViewModel tervezési mintát használva.

Model-View-ViewModel – MVVM

- A **View** tud a **ViewModel**-ről.
- A **ViewModel** tud a **Model**-ről.
- A **Model** nem ismeri a **ViewModel**-t és a **ViewModel** nem tud a **View-ról**.

ViewModel

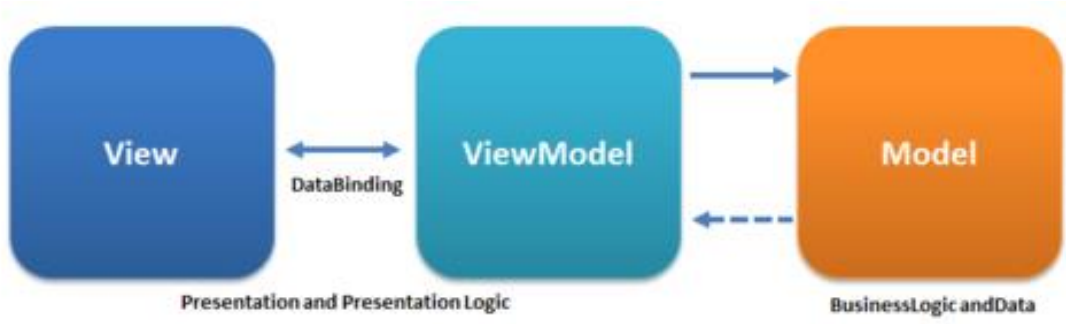
- Olyan tulajdonságokat és parancsokat implementál, amikhez a **View** adatkötést (Binding) végezhet.
- A változásértesítési eseményeken keresztül értesíti az állapotváltozások nézetét.

MVVM előnyök és hátrányok

- **Előnyök**
  - Tesztelés lehetősége
  - Bővíthetőség
  - Karbantarthatóság
- **Hátrányok**
  - A sok binding miatt bonyolult a debuggolás.
  - Egyes esetekben nehéz megtervezni egy komplexebb ViewModel-t.

Felhasználói felület alkalmazása

1. Létrehozunk egy **Model**-t, ami tartalmazza az alkalmazás logikáját és az adatmodellt.
2. Létrehozunk egy **View**-t, ami megjeleníti az adatokat, amiket a **Model** tartalmaz.
   a. Kezelheti a felhasználói interakciókat is.
3. Létrehozunk egy **ViewModel**-t, ami kapcsolódik a **Model**-hez és a **View**-hoz.
4. Összekötjük a **View**-t és a **ViewModel**-t adatkötés segítségével.
   a. Az adatkötés teszi lehetővé a kommunikációt a két réteg között.



Felhasználói felület alkalmazása a Model-View-Controller tervezési mintát használva.

Model-View-Controller – MVC

- A **View** réteg frissíti a **Controller** réteget.
- A **Controller** réteg frissíti a **Model**-t.
- A **Model** réteg közvetlenül visszahat a **View** rétegre.
- Ismertebb MVC web keretrendszerek
    - Ruby on Rails, Django, ASP.NET, Symphony

Controller

- A **Controller** felelős azért, hogy értelmezze a felhasználói interakciókat és módosítsa a **Model**-t ennek megfelelően.

MVC előnyök és hátrányok

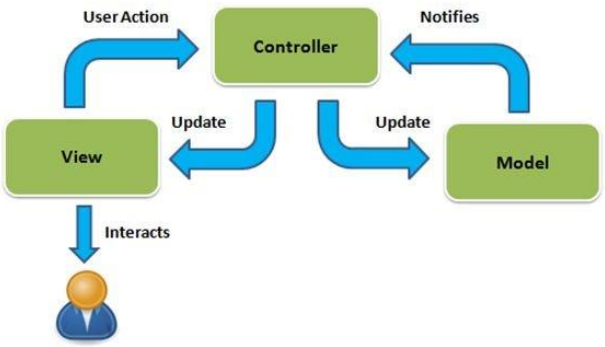
- **Előnyök**
    - Közös munkát lehetővé teszi a többi fejlesztő számára.
    - Hibakeresés viszonylag könnyebb, mert több szint van.
    - Tesztelhető minden komponens külön-külön.
- **Hátrányok**
    - Nehezen újrahasználhatóak a modellek.
    - Fejlesztés során több technológia ismeretére lehet szükség.

Felhasználói felület alkalmazása

- Felhasználói felület elkészítésekor a felhasználói interakciókat és azok hatását a **Model**-re és a **Controller**-re kell szétválasztani.

MVC folyamata

1. Felhasználó csinál valamit a UI-on (**View**).
2. A **View** tájékoztatja a **Controller**-t a végrehajtott műveletről.
3. A **Controller** frissíti a **Model**-t.
4. A **Model** új adatokat szolgáltat.
5. **Controller** frissíti a **View**-t.



Szoftver management

A fejlesztési folyamat részei

Programkészítés lépései

1. **Specifikáció**
    a. A megrendelő kitalálja, hogy milyen programot szeretne.
    b. Megbeszéli a programozóval, hogy mit kell tudnia a programnak.
    c. **A tisztázott igények precíz leírását nevezzük specifikációnak.**
2. **Tervezés**
    a. A feladathoz megfelelő adatszerkezeteket és algoritmusokat kell találni vagy fejleszteni.
    b. Megtervezendő továbbá a program felhasználói felülete és gondolni kell a jövőbeli bővíthetőségre is.
3. **Kódolás**
    a. A kész terveket egy választott programozási nyelven kódoljuk.
    b. A forráskódból egy fordítóprogram hozza létre a futtatható programot.
4. **Tesztelés**
    a. Két szempont alapján kell tesztelni a programot:
        i. **Minden esetben működik-e és hatékony-e.**
5. **Hibajavítás**
    a. A felismert hibák kijavításához új specifikáció, új terv és újrakódolás lehet szükséges.
6. **Dokumentáció**
    a. Ha később is szeretnénk fejleszteni a programot és érteni akarjuk a működését, akkor egy dokumentációt kell készíteni hozzá.
    b. Dokumentációnak **két szintje** van:
        i. **Felhasználói** és **Fejlesztői**

Szoftvertervezés egy lehetséges menete

1. **Deployment diagrammok**
    a. A rendszer és milyen körülmények között lesz használva.
2. **Behavioral diagrammok**
    a. Milyen funkciókat kell a rendszernek tudnia?
    b. Use-case + Activity + Wireframes diagrammok
    c. Megrendelővel közös tervezés
3. **Structural diagrammok**
    a. A működést milyen modulokkal, milyen felbontással lehet megoldani.
    b. Component + Sequence (Class diagrammok)
    c. Entity-relations diagram (Adatbázis tábla struktúrák)
4. **Időtervezés**
    a. Gantt diagram
5. **Implementálás**

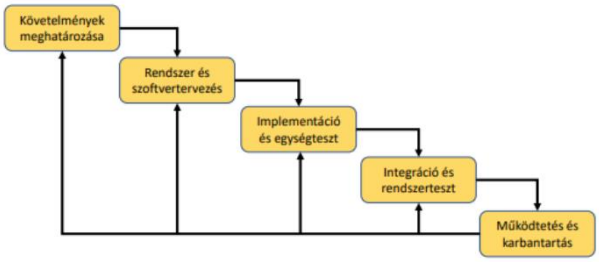
Fejlesztési folyamat közbeni problémák

- **Követelmények változása:** Ügyfelek igényei fejlesztéskor folyamatosan változhatnak.
- **Hiányos tesztelés:** Hibás alkalmazásokat vagy váratlan problémákat okozhatnak.
- **Hibás kommunikáció**
    - Fejlesztők és az ügyfelek között, de lehet a fejlesztői csapaton belül is rossz kommunikáció.
- **Technikai korlátok**
    - Akadályozhatja a fejlesztés menetét, minőségét az, hogy nem állnak rendelkezésre a megfelelő eszközök, technológiák a fejlesztés során.

Klasszikus modellek

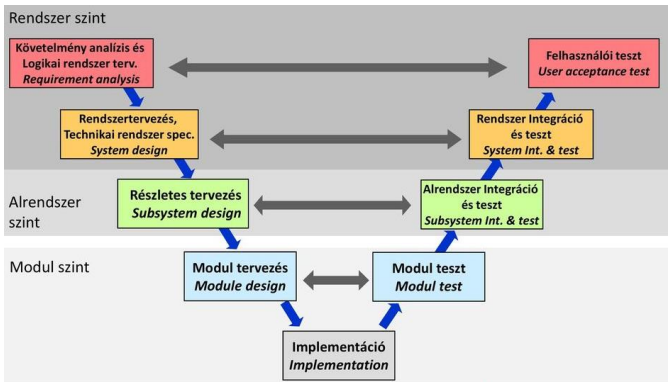
Vízesés modell

- Akkor hasznos, ha a követelmények jól ismertek és csak nagyon kis változások lehetségesek a fejlesztéskor.
    - Kevés üzleti rendszernek vannak stabil követelményei.
    - Főleg nagy rendszerek fejlesztésekor használják, ahol a fejlesztés több helyszínen történik.
- **Problémái**
    - Minden a specifikáció minőségétől függ.
    - Későn lát a megrendelő működő programot.
    - Kezdeti bizonytalanságot nehezen kezeli.
    - Tesztelés szerepe nem eléggé hangsúlyos.
- **Fázisai**



V-modell

- Azért nevezik V-modellnek, mert két szára van: **Fejlesztési** és **tesztelési** szár.
- Vízesés modell kiegészítése teszteléssel.
    - Először végre kell hajtani a fejlesztés lépéseit, ezután jönnek a tesztelés lépései.
    - Ha valamelyik teszt hibát talál, akkor vissza kell menni a megfelelő fejlesztési lépésre.
- Szigorú dokumentálást követe és nem küszöböli ki a vízesés modell problémáit.



Agilis módszerek

Scrum

- **Vezetőség az erőforrásokat adja**: Emberi, anyagi és infrastrukturális
- Adott Scrum Team közvetlenül kommunikál a megrendelővel.
- Teljesen az ő felelősségük a projekt esetleges kudarca.

Scrum szerepkörök

- **Product Owner**
  - Csapat tagja, megrendelő érdekeit képviseli.
  - Megszervezi a rendszeres demókat.
  - Nem fejlesztő, de érti a gazdasági folyamatokat.
  - Product Backlog kezelője.
- **Scrum Master**
  - Csapat tagja, egy hagyományos projektmenedzser szereppel egyezik a feladata.
  - Felügyeli a folyamatokat.
  - Konfliktusokat kezel, akadályok elhárítását irányítja.
  - Meetingeket ő szervezi, ő vezeti.
  - Scrum csapat feje.
- **Scrum Team**
  - 5-9 fő alkotja, **szükséges:** Elemző, fejlesztő, tesztelő
  - Ők végzik a tényleges fejlesztést.
  - Felelősségük, hogy egy sprintre bevállalt feladatokat elvégezzék.
  - **Fejlesztői fokozatok**: Junior > Medior > Senior
  - **Tesztelő**
    - Mindenki a saját kódját teszteli.
    - Unit tesztek után peer review más fejlesztővel.
    - Van manuális teszt, amit nem a fejlesztők végeznek.

Scrum entitások

- **User Story, Task**
  - Specifikációból eredő feladat.
  - Taskokra bomlik szét, ezeket veszik magukra a fejlesztők.
- **Sprint**
  - 1-4 hét hosszú fejlesztési szakasz.
  - Addig jönnek újabb sprintek, amíg a Product Backlogból el nem tűnnek a User Storyk.
  - A sprint vége egy kész szoftver.
- **Product Backlog**
  - Elkészítésre váró Story-k gyűjtőhelye.
  - Product Owner tartja karban, tehát prioritásokat rendel a story-khoz.
  - **ROI:** Return of Investment = üzleti érték / ráfordítás.
- **Sprint Backlog**
  - A Product Backlog, de csak az adott sprintre bevállalt storykra szűrve.
- **Burn down/up chart**
  - Napi eredmény diagram és megmutatja, hogy a csapat mennyire tartja az eredeti ütemtervet.
- **Impediment**
  - Akadály, ami a munkát hátráltatja, valamilyen munkahelyi probléma.
  - Scrum Master feladata eljárítani.

Indító meetingek

- **Sprint (pre)grooming**
  - Architektek + Scrum Master
  - Storyk ellenőrzése, pontozása (nehézség/idő > Fibonacci számokkal)
- **Sprint planning**
  - Team bevállal storykat.
  - Figyelembe veszik a pontozást.

Folyamatos meetingek

- **Daily stand-up**
  - Minden nap ugyanakkor és az egész team részt vesz benne.
  - Szóban, kb 15 perc és egyéb eszközök nélkül.
  - Megbeszélik, hogy ki mit csinált vagy ma mit fog csinálni, és kinek mi a problémája.
- **Sprint refinement**
  - Hetente 1 maximum (3 hetes sprintben maximum 2 alkalom)
  - Storyk/Taskok áttekintése
  - Gyors hibaelhárítások
  - Nem része a Scrumnak, de alkalmazzák.
- **Egyéb meeting**
  - Csapat részhalmaza egyeztet.

Lezáró meetingek

- **Sprint review**
  - Team + Scrum Master + Product Owner
  - Eredmények bemutatása, Product Owner dönti el, hogy sikeres-e a sprint.
  - Kimaradt Storyk/Taskok a következő sprintre mennek.
- **Sprint retrospect**
  - Review után tartják, Team + Scrum Master
  - Személyes tapasztalatok/javaslatok megvitatása.
  - Céges szintű problémákat a Scrum Master továbbítja felfelé.
  - Személyes konfliktusok megbeszélése.

Kanban (jelzőtábla vagy hirdetőtábla.

- Lean módszer emberek csoportos munkájának menedzselésére és fejlesztésére.
  - **Lean**: Vállalatirányítási módszer, aminek célja, hogy a vállalat minél gazdaságosabban állítsa elő a termékeit, szolgáltatásait.
- Kiegyensúlyozza az igényeket és elérhető munkakapacitást.
- Munkafolyamat megjelenítésére használják a Kanban táblát.
- A munkafolyamatok elvégzését úgy ütemezik, ahogy a dolgozók kapacitása megengedi és nem engedik, hogy a munkafolyamatok sürgőssége határozza meg a munka időbeosztását.

Kanban célja

- Célja, hogy létrejöjjön egy olyan vizuális folyamatmenedzsment-rendszer, ami segít meghozni azokat a döntéseket, hogy mit, mikor és hogyan gyártsanak.

Hol használják?

- Maintenance/support jellegű feladatoknál
- HR-folyamatokra
- Értékesítésre

A verziókövetés

Alapfogalmak

Repository

- **Adatbázis neve**, ami a **verziókezelt fájlokat tárolja**.
- Egy mappa fájlokkal, amiben egy speciális rejtett **.git** mappa, ami a kezeléshez szükséges adatbázist tárolja.

Commit

- A fájlokról elmentett pillanatkép, amihez egy kommentet írunk, hogy éppen mit csináltunk.

Server/Origin

- Szerver, ami a **repository**-t tárolja és kiszolgálja a fejlesztőknek.
- A fejlesztők a saját gépükön rendelkezhetnek privát repository-val is, de a többi fejlesztőnek csak az lesz elérhető, ami a szerveren van.
- Mivel a Git elosztott rendszerű, ezért nem szükséges a repository-t publikálni egy másik szerverre.

Client

- Kliens gép, ami a szerverhez csatlakozik.

Working Set/Working Copy

- A fejlesztő helyi gépén tárolt változata a repository-nak.

Revision number

- Repository-ban tárolt változatokra ezzel tudunk hivatkozni.
- Git-ben ez egy hasító függvénnyel képzett érték, ami egyedileg és egyértelműen beazonosítja a változatokat.

Head

- Legutolsó revision number hivatkozó kifejezése.

Main/branch

- A main a fő ág, ebből az ágból bármelyik ponton készíthető al ág, amit branch-nek nevezünk.
- A branch létrehozása után külön változatként él tovább.
- A létrehozott branch-ek bármikor beintegrálhatóak bármelyik ágba, akár vissza a fő ágba is.
- Előnyük, hogy kísérletezhetünk vele, mint például új funkciók fejlesztésére.

Merge

- Két ág, branch összeolvasztásának folyamata.
- Általában nem igényel manuális beavatkozást.

Push

- Lokális változások feltöltése távoli repository-ba.

Tag

- Címkézés, például programverzió.

Fork

- Szerveren megosztott repository helyi másolata, amin mi dolgozhatunk.

Pull request

- Távoli repository változásainak letöltése és mergelése.

Diff

- Változások megtekintése

Reset

- Nem commitolt módosítás eldobása.

Stash

- Módosítás mentése átmeneti tárolóba.

Checkout

- Branchek közötti váltás.

Fetch

- Távoli repository változásainak letöltése.

A GIT sajátosságai

.gitignore

- Ez egy rejtett dot file, ami nem kerül feltöltésre a repository-ba, csak lokálisan van a fejlesztő gépén.
- Ebben a fájlban megadhatunk „szabályokat”, amivel megmondhatjuk, hogy mit ne töltsön fel.
- Például dot fájlok, konfig fájlok, csomagok.

Támogatottság

- Különböző operációs rendszerre és fejlesztői eszközre kiterjed, például fejlesztői környezetekben is megtalálhatóak, mint például a Visual Studio, Jetbrains, stb.

Problémák/konfliktusok kezelése

- Konfliktus akkor keletkezik, amikor a verziókezelő rendszer nem tud dönteni a változások sorsáról.
- Akkor következhet be, amikor egyszerre többen dolgoznak ugyanazon a projekten.
- Ilyenkor manuálisan kell megoldani a konfliktusokat, hogy melyik sor az, ami kell.
- A konfliktusokat parancssoron és erre kialakított szoftvereket is tudunk használni:
    - GitKraken
    - Github Desktop
    - Sourcetree

Fejlesztési modellek (Gitflow, Trunk-Based, Linux Kernel)

Gitflow

- Git munkafolyamat, ami lehetővé teszi a fejlesztők számára, hogy szabályozzák a fejlesztési folyamatot különböző ágak között.
- **Fő elemei:**
    - **Master ág:** Tartalmazza a stabil, befejezett és tesztelt kódokat.
    - **Develop ág:**
        - Minden új funkció, javítás vagy változtatás a develop ágban indul, majd a tesztelés után összeolvasztják a master ággal.
    - **Feature ág:**
        - Új funkciók a develop ágba kerülnek.
        - **Soha nem kerül interakcióba a main ággal.**
    - **Release ág:**
        - Új verzió előkészítésének ága, ami kiadásra van szánva, amit mergelünk a main ággal és egy verziószámmal látjuk el.
    - **Hotfix ág:**
        - Hibajavításokat tartalmaz, amik összeolvasztásra kerülnek a master ággal, majd a develop ággal.

Gitflow folyamata

1. Létrehozunk egy develop ágat a main ágból.
2. Létrehozunk egy release ágat a develop ágból.
3. Létrehozunk egy feature ágat a release ágból.
4. Ha egy funkció elkészült, akkor a develop ágba mergelődik.
5. Amikor a release ág elkészült, akkor mergelődik a develop és a main ággal.
6. Ha a main ágban hibát találunk, akkor a main ágból létrehozunk egy hotfix ágat.
7. Ha a hotfix elkészült, akkor mergelődik a develop és a main ágba.

Trunk-Based

- Minden fejlesztő lokálisan és önállóan dolgozik a projektjén, majd a változásait legalább naponta egyszer mergeli a main ágba.
- A mergenek függetlenül attól kell történnie, hogy a funkcióváltások vagy kiegészítések befejeződtek-e vagy sem.
- Kevesebb ágat tartalmaznak és kevesebb a merge conflict.
- Alapfeltétele a CI/CD, hogy automatizálva legyen minden.

Gitflow vs Trunk-based

- Gitflow-t érdemes komplexebb projektekben használni, ahol több ág van.
- Trunk-based kisebb team számára kedvezőbb.
- Összességében a választás függ a projekttől, team-től.

Linux Kernel

- Fejlesztők közös fejlesztési erőfeszítéseinek eredménye.
- Kis létszámú maintainerekből álló team, akik felügyelik a hozzájárulásukat és biztosítják, hogy azok összhangban legyenek a projekt általános céljaival.
- **Linux Kernel folyamata:**
    - **Ötlet/Idea:** A fejlesztőnek támad egy ötlete.
    - **Fejlesztés/Development:** Lefejleszt egy új funkciót és azt alaposan teszteli is.
    - **Benyújtás/Submission:** A fejlesztő benyújtja a kódot, hogy a maintainer felülvizsgálja.
    - **Értékelés/Review:** A maintainer ad egy visszajelzést, hogy esetleg min kellene még változtatni, amíg az nem felel meg.
    - **Tesztelés/Testing:** A maintainer és más fejlesztők is tesztelik a kódot, hogy biztosan stabil-e.
    - **Integráció/Integration:** Ha a kód átment a teszten és a maintainer elfogadta, akkor beintegrálják a kernel kódbázisába.
    - **Release:** Új kernel verzió nyilvánosságra kerül és ezzel a ciklus kezdődik elölről.

UML (Unified Modeling Language)

UML nélkül

- **Kommunikációs szakadék van a megrendelő és a fejlesztők között.**
    - Agilis módszereket be kell vonni.
    - Prototípusokat kell fejleszteni és azokat véleményeztetni.
    - Meg kell találni a közös nyelvet.
- **Kommunikációs szakadék van fejlesztő és fejlesztő között**
    - Tapasztalat, tudásszint béli különbségek lehetnek.

Alapelvek

- Grafikus leírónyelv, ami segít vizualizálni, specifikálni, tervezni és dokumentálni.
- **Kinek jó?**
    - Megrendelő egy folyamatábrát könnyen tud értelmezni.
    - Fejlesztő könnyebben megérti, hogy a másik fejlesztő rendszere hogy működik.
    - Vizuális ábrázolás jobb megértést biztosít.
    - Dokumentáció és így alapos lesz általa.

UML használata

- UML egy szigorú modellező nyelv.
- **Modellező eszközök:**
    - Microsoft Visio
    - Visual Paradigm for UML
    - Rational Rose

Célok

- Egyszerűsíti a bonyolult struktúrákat.
- Kommunikációs eszközként szolgál.
- Automatizálja a szoftverek előállítását és folyamatokat.
- Segíti a szerkezeti problémák megoldását.
- Javítja a munka minőségét.

Diagramok bemutatása (kategóriák: Deployment / Behavioral / Structural / Implementation).

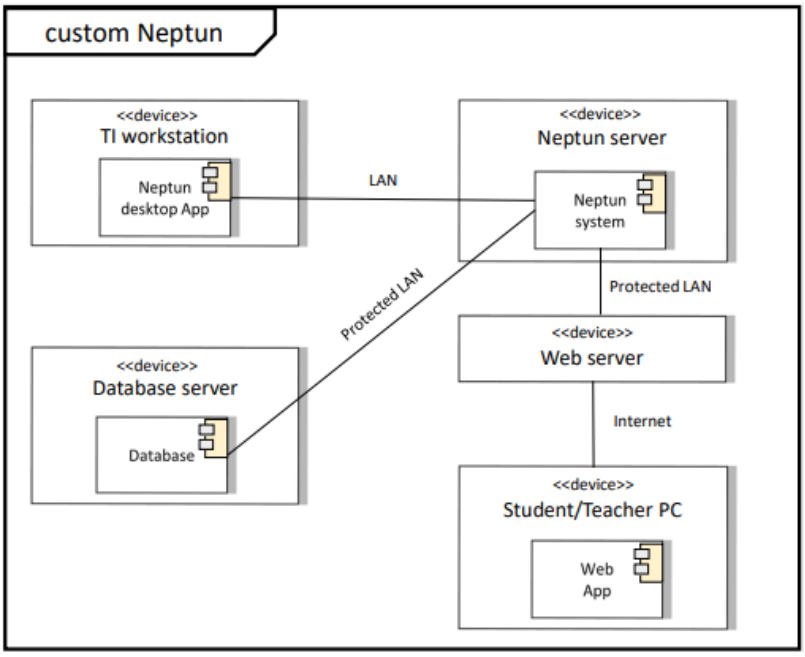
Structural

Structural – Composit Structure Diagram

- A Composite Structure diagram egy UML diagram, ami megmutatja a szoftverrendszer belső szerkezetét.
- Tartalmazza az osztályokat, interfészeket, csomagokat és azok kapcsolatait.
- Segít a felhasználónak látni, hogy mi van egy objektumon belül és hogyan illeszkednek össze a különböző tulajdonságok.
- A részek (**parts**) a strukturált osztályozó (**classifier**) által tulajdonolt egy vagy több példányt jelentik.
- A csatlakozók (**connectors**) a részek közötti kommunikációt jelölik.
- A portok (**ports**) az osztályozó példány és annak környezete közötti interakciós pontokat jelölik.
- A kollaboráció (**collaboration**) az osztályozók közötti interakciók leírására szolgál.

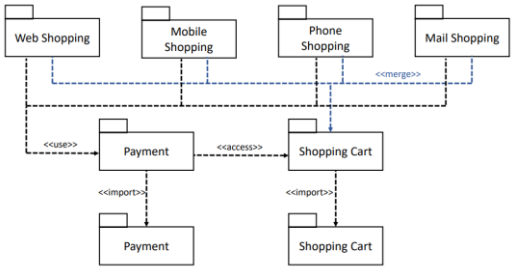
Structural – Deployment Diagram

- A rendszer futásának elemeit bemutató diagram.
- **A működtető elemek lehetnek**
    - Számítógépek
    - Hálózati csomópontok
    - Egyéb környezetek (VM, konténer)
- **Akár a fejlesztési fázis első diagramja is lehet**
    - Ha a környezet már készen van (új szoftvert kell írni meglévő környezetre)
    - **Új rendszernél a részletes tervezéskor használjuk.**



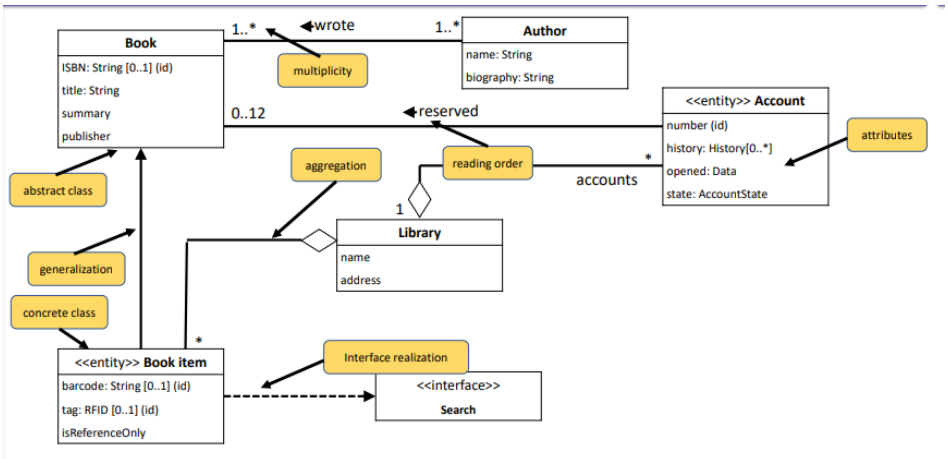
Structural – Package Diagram

- **Package diagrammon ábrázoljuk a különböző DLL-eket, rétegeket és a köztük való összevonásokat, projekt függőségeket.**
- Felépítés, tartalmazás, függőség (Amiket csomagolni lehet pl. egy DLL-be)
- **Kapcsolatok**
    - **package**: Maga a Class Library (DLL)
    - **package merge**: <<merge>>
    - **usage dependency**: <<use>>
        - Függőség, mert lehet, hogy egy másik package is használhatja
    - **private import:** <<access>>
        - Projekt referenciaként használ egy másik package-t.
        - Szigorúbb kapcsolat
    - **public import:** <<import>>
        - Lazább kapcsolat



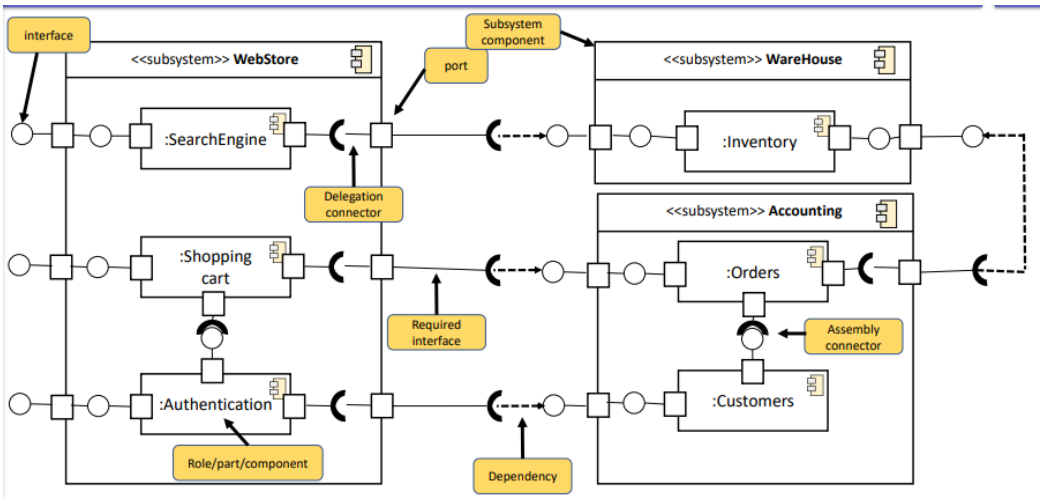
Structural - Class Diagram

- Osztályok vagy konkrét objektumok ábrázolása
    - Adattagok, metódusok
- Korai tervezéskor még nem készül el, ilyenkor a Package diagram interfészei elegek.
- **Multiplicitás**
- **Aggregáció**
- **Reading order**
- **Generalization**
- **Concrete class**
- **Interface realization**
- **Attributes**



Structure – Component Diagram

- Alrendszereket tartalmaz, azon belül milyen osztályok vannak és ezeknek milyen kapcsolataik vannak.
- Abban különbözik a Package diagramtól, hogy külső szolgáltatásokat is jelölhetünk benne.
- **Subsystem component:** Alrendszer
- **Component:** Komponens, részek
- **Interface:** körökkel ábrázoljuk
- **Port:** Négyzettel ábrázoljuk, várunk paramétert konstruktoron interfésszel
- **Delegation connector:** Kapcsolat
- **Required interface:** Vár interfészt
- **Assembly connector:** Alrendszerekben komponensek közötti kapcsolat
- **Dependency:** Függőség, Dependency Injection



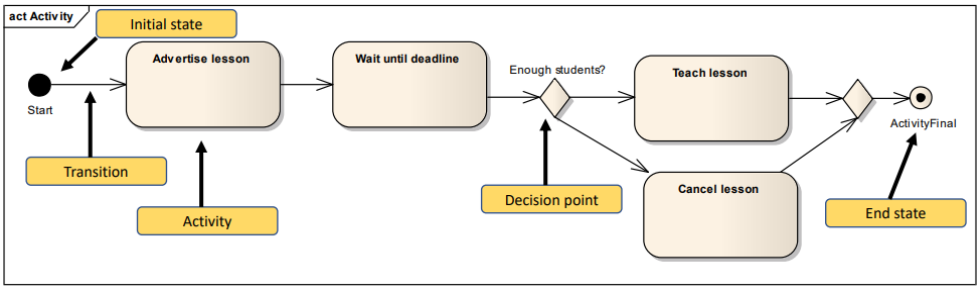
Behavioral

Behavioral – Use Case Diagram

- **Célja:** Megrendelővel való egyeztetés, hogy pontosan milyen szerepköröket, funkciókat képzelt el, és ezeket a funkciókat melyik szerepkörrel lehet igénybe venni.
- **Aktor és használati eset közötti megfeleltetés.**
- **Van öröklődés:**
  - Aktorok között
  - Használati esetek között
- **Tartalmazás/Kibővítés**
  - Használati esetek között

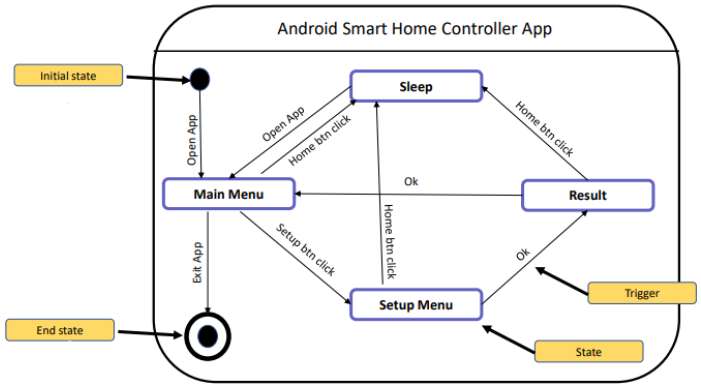
Behavioral – Activity Diagram

- Use-case utáni következő lépés.
- **Célja**:
  - Rendszer folyamat lerajzolása.
  - Leírja az egyik tevékenységtől a másikig tartó sorrendet.
  - Leírja a rendszer párhuzamos, elágazó és egyidejű folyamatát.
- A lépésenkénti tevékenységek és műveletek munkafolyamatainak grafikus ábrázolása.
- **Használati eseteknél a cél:**
  - Belső folyamatok ábrázolása
  - Egymás után következőségek ábrázolása
- 1 használati eset = 1 Activity diagram
- Egy diagrammon belül más use case-ek is előjöhetnek.
- **Initial state, End state:** kezdő és vég állapot
- **Transition:** Átmenet
- **Activity:** Aktivitás
- **Decision point:** Döntési pontok



Behavioral – State Machine Diagram

- Rendszer/Objektum állapotainak egymás után következőségét ábrázolja.
- Irányított gráf, aminek csomópontjai a logikai állapotok, amik élei a köztük lévő átmenetek.
- A végrehajtható műveletek az állapotokhoz és az átmenetekhez is tartozhatnak.
- **Probléma, hogy egy idő után átláthatatlan lesz a diagram.**
    - State transition
- **Initial, end state:** kezdő és vég állapot
- **State:** az adott állapot
- **Trigger:** átviszi egyik állapotból a másikba

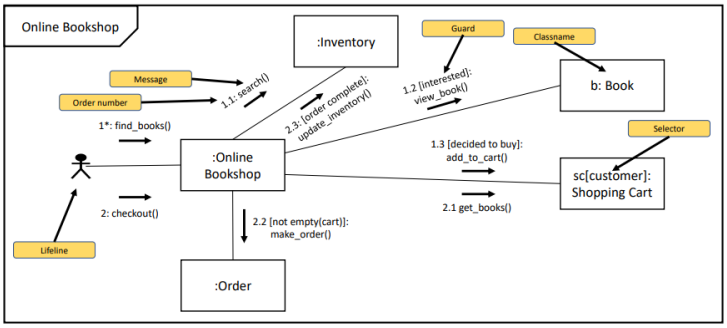


Behavioral – Interaction Diagram

- Egy folyamat résztvevői (aktorok vagy modulok vagy rétegek) közötti kommunikációs folyamatokat ábrázoljuk
- Három különböző diagramot különböztet meg:
    - **Communication diagram:** résztvevők és sorrend
    - **Timing diagram:** időzítési információk, megkötések és állapotok is
    - **Sequence diagram:** ciklusok, feltételek és élettartamok is.
- Mindhárom típus ugyanarra a célra való csak más mélységben mutatja be a kommunikációs folyamatokat.

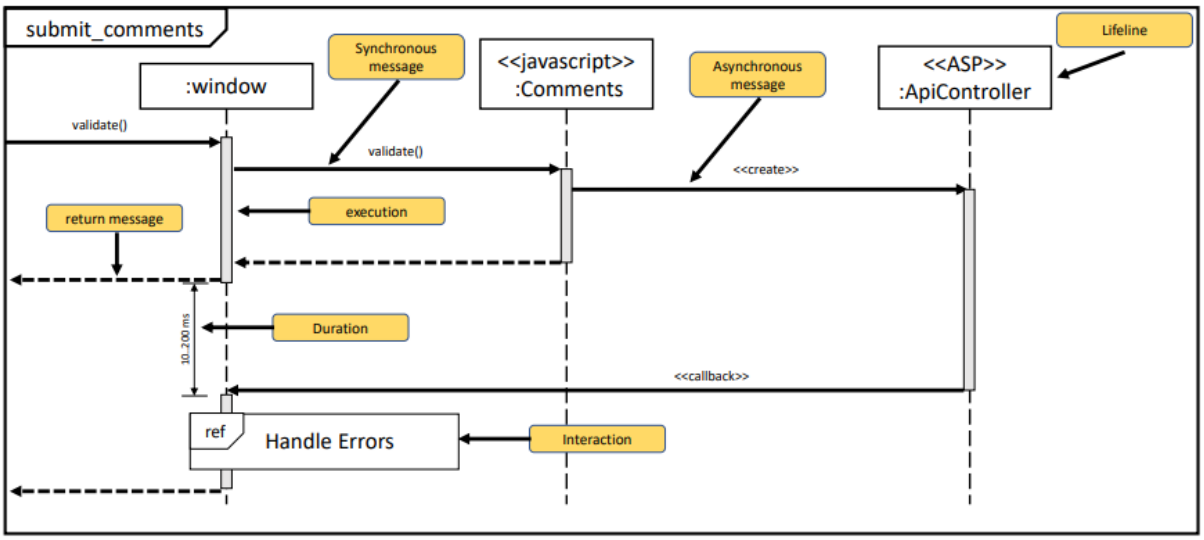
Behavioral – Communication Diagram

- **Lifeline:** Felhasználó hogyan tud az osztályokkal kommunikációba kerülni.
- **Message:** Vmilyen metódus végrehajtása
- **Order number:** Sorrendet írja le
- **Guard:** „Őrszem”, csak akkor hajtódjon végre ha xy
- **Classname:** Osztály neve
- **Selector:** Paramétereket adhatunk át, jelzésre jó



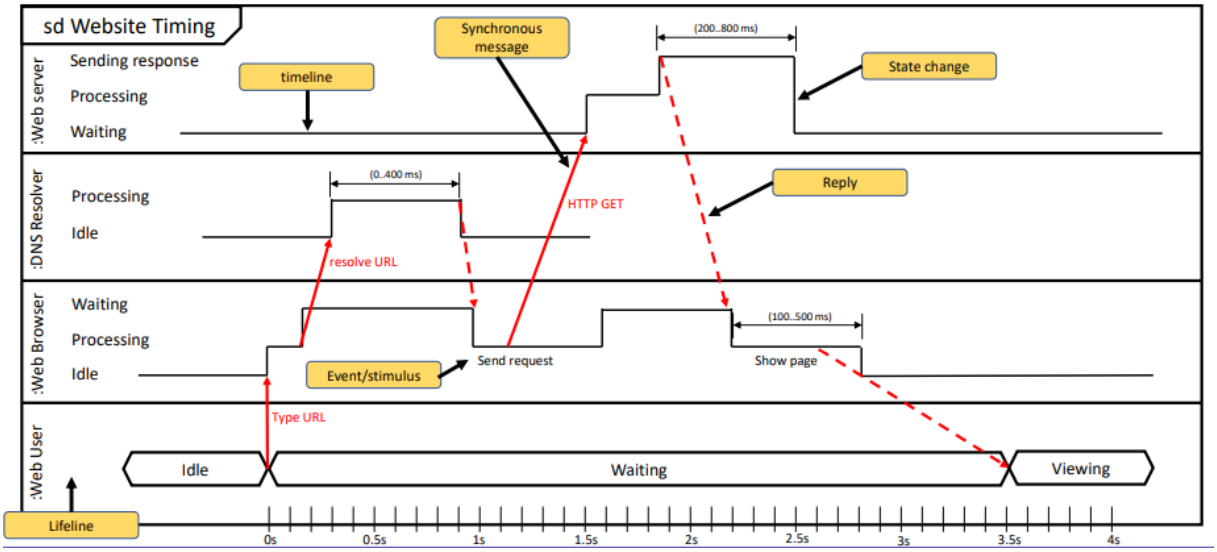
Behavioral – Sequence Diagram

- Validációk, majd azoknak válasza, de validáció közben végrehajtódhat egy másik folyamat.
- **Lifeline:** Pl egy ApiController
- **Synchronous message:** Visszaad üzenetet
- **Asynchronous message:** Nem ad vissza üzenetet
- **return message:** Vissza ad egy értéket vagy üzenetet



Behavioral – Timing Diagram

- **Lifeline:** „Életvonal”
- **Timeline:** Idővonal", ami az lifeline-on belül mutatja, hogy az ő timeline-ja az a különböző állapotok között hogyan változik.
- **Synchronous message:** Kérést küldünk az egyik lifeline-ból a másikba egy kérést, és az időzítések miatt nincsenek **asynchronous message**-k.
- **Reply:** Válasz
- **State change:** Függőleges vonalak az állapot változások.
- **Event/stimulus:** Címke, ami vmilyen triggert jelöl.

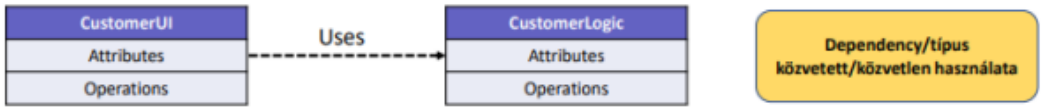


Gang-of-Four tervezési minták 1

OO relációk

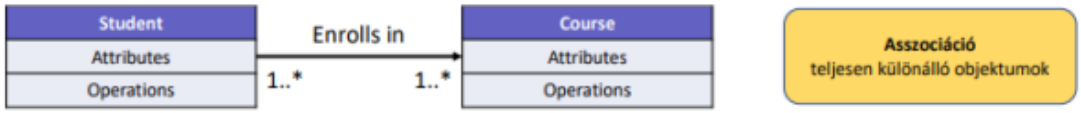
Dependency - Függőség

- Két elem között akkor áll fenn, ha az egyik (a független) elem változása hatással van a másik (a függő) elemre.
- Kölcsönös függőség akkor van, ha mindegyik elem hat a másikra.



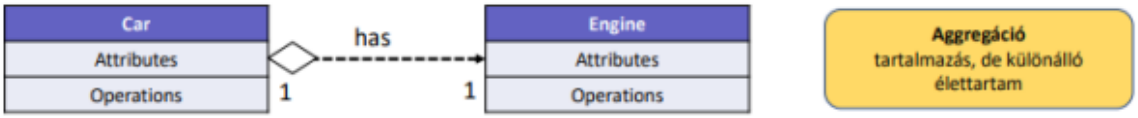
Asszociáció

- Osztályok közötti tetszőleges viszony.
- Asszociációs kapcsolat áll fenn két osztály között, ha az egyiknek a saját helyes működéséhez ismernie kell a másikat.
    - Az egyik használja a másikat.
    - Az egyik tartalmazza a másikat.



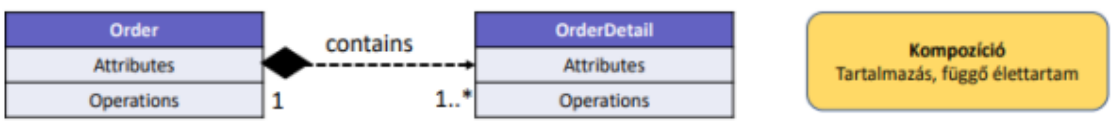
Aggregáció

- Az asszociáció speciális esete, tartalmazási kapcsolat.
- A tartalmazó osztály példányai magukba foglalják a tartalmazott osztály egy vagy több példányát, ez a rész-egész kapcsolat.
- A tartalmazó és a tartalmazott osztály egymástól függetlenül létezhetnek.
- A tartalmazás lehet, erős illetve gyenge.
- **Erős aggregáció:** A részek élettartalma szigorúan megegyezik az egészével, ez a kompocízió.
- **Gyenge aggregáció:** Egyszerű/általános aggregáció.



Kompozíció

- Másnéven **erős aggregáció**, tehát szigorú tartalmazási kapcsolat.
- Egy rész objektum csak egy egészhez tartozhat.
- **A tartalmazó és a tartalmazott életciklusa közös**: Például van egy User objektum, aminek van egy Address tulajdonsága. Ha a User objektum megszűnik, akkor megszűnik az Address is, de nem létezhet User objektum Address nélkül. Ezért közös az életciklusuk.



Újrahasznosítható kód fogalma

- Időt spórolhatunk vele, mert kiküszöböli a már tökéletesen működő és használható kódrészletek újraírásának szükségességét.
- A hasonló funkciók kódját gyakran több projektben is fel lehet használni, hogy felgyorsítsuk a fejlesztés menetét.
- Kockázatok csökkentése azáltal, hogy egy már kipróbált, tesztelt kódot használtunk fel.
    - Ez garantálhatja a jó felhasználói élményt, így zökkenőmentesen is működhet.
- Megakadályozza a kód rohamos növekedését, ami lassú, erőforrás igényes is lehet.
    - Törölni kell a már nem használt kódrészleteket.
- **Nehézségek:**
    - Kommunikáció a projekt növekedése miatt.
    - Dokumentáció készítése, hogy például az adott kódrészletet hol lehet felhasználni újra.

Kompozíció és öröklés összehasonlítása

	Öröklés	**Kompozíció**
Kapcsolat	Az autó egy jármű. (x egy y)	Az autónak van kormánykereke. (x-nek van y-ja)
Cél	Az öröklésnek a célja, hogy megtervezzük, hogy az adott osztály mi az	A kompozíciónak a célja, hogy az adott osztály mit csináljon.
Egyéb	Sokkal szorosabb a kapcsolat az objektumok között	Sokkal lazább a kapcsolata
	Az ősosztály módosítása kihatással lehet a leszármazottakra.	Rugalmasabb, mert futásidőben tudunk módosítani.

SOLID elvek

Single Responsibility

- Minden osztály egy dologért legyen felelős és azt jól lássa el.
- **Ha nem követjük, akkor:**
    - Spagetti kód, átláthatatlanság
    - Nagy méretű objektumok
    - Mindenért felelős alkalmazások és szolgáltatások

Open/Closed elv

- Egy osztály legyen nyitott a bővítésre és a zárt módosításra, vagyis nem írhatunk bele, de származtathatunk tőle.
- **Ha nem követjük, akkor:**
    - Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetőek.
    - Leszármazott megírásakor módosítani kell az ősosztályt, ami tilos.
    - Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában.

Liskov substitutable

- Ősosztály helyett utódpéldány legyen mindig használható.
- Compiler supported, hiszen OOP elv (polimorfizmus)
- Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.

Interface seggregation

- Sok kis interfészt használjunk egy hatalmas mindent előíró interfész helyett.
- **Ha nem követjük, akkor:**
    - Egy osztályt létrehozunk valamilyen célból, megvalósítjuk az interfészt és rengeteg üres, fölösleges metódusunk lesz.
    - Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással.

Dependency Inversion

- A függőségeket ne az őket felhasználó osztály hozza létre.
- Várjuk kívülről a példányokat interfészeken keresztül.
- **Példány megadására több módszer is lehetséges:**
    - Dependency Injection
    - Inversion of Control (IoC) container
    - Factory tervezési minta
- **Ha nem követjük, akkor**
    - Egymástól szorosan függő osztályok végtelen láncolata.
    - Nem lehet modularizálni és rétegezni.
    - Kód újrahasznosítás lehetetlen

Egyéb elvek

- DRY, Don’t Repeat Yourself
- DDD = Domain Driven Design

Gang-of-Four tervezési minták 2

Dependency inversion módszerek

- A függőségeket ne az őket felhasználó osztály hozza létre.
- Várjuk kívülről a példányokat interfészeken keresztül.
- Példány megadására több módszer is lehetséges
    - Dependency Injection
    - Inversion of Control (IoC) container
    - Factory tervezési minta
- **Ha nem követjük, akkor**
    - Egymástól szorosan függő osztályok végtelen láncolata
    - Nem lehet modularizálni és rétegezni
    - Kód újrahasznosítás lehetetlen

Dependency Injection

- Lazán csatoltság kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.
- **Probléma**
    - A kód függjön absztrakciótól, ne konkrét implementációtól.
- **Megoldás**
    - Az interfészt várjuk paraméterként a konstruktorban.
    - Setter injektálás, amikor az objektumokat setter metódusok segítségével injektáljuk.

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 principle
- **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.

IoC minták

- Dependency Injection
- Observer Pattern
- Template Method

Observer (Behavioral)

- Hogyan tudják az objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól.
- **Az Observer az egyik leggyakrabban használt minta!**
- **Probléma**
  - Vevő szeretne vásárolni egy új terméket, de nem szeretne mindennap meglátogatni az üzletet, ahol lehet kapni.
  - Az üzlet pedig nem szeretné feleslegesen fogyasztani az erőforrásait abból a szempontból, hogy minden egyes új termék miatt küldözget emailt, mert ez csak spam lenne.
  - Tehát a vevő pazarolja a saját idejét vagy az üzlet az erőforrásait pazarolja.
- **Megoldás**

- Kell egy subscriber, amivel feliratkozunk valamire és az értesít.
- Feliratkozó osztályok megvalósítanak egy ISubscriber interfészt.
- Írjon elő egy StateChange() vagy Update() metódust.
- A subject kezelje a feliratkozókat Subscribe(), UnSubscribe()
- Állapotváltozáskor hívja meg az összes feliratkozó StateChange() metódusát.
- A feliratkozók tegyék meg a frissítési lépéseket.

Observer használjuk, ha

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

Observer implementálása

1. Business logic két részre bontása:
    a. Alapvető, más kódtól független funkcinalitás fog publisher-ként működni.
    b. A többi pedig subscriber osztályok halmaza lesz.
2. Subscriber interfész deklarálása és legalább egy frissítési metódust kell deklarálnia.
3. Publisher interfész deklarálása, subscriber-ben implementáljuk ezeket a metódusokat.
    a. A publisher-ek csak a subscriber-ekkel dolgozhatnak a subscriber interfészen keresztül.
4. Hozzunk létre egy absztrakt osztályt, ami közvetlenül a publisher interfészből származik.
    a. A publisher-ek kiterjesztik ezt az osztályt, örökölve a subscriber behavior-t.
5. Publisher osztályok létrehozása.
    a. Minden alkalommal, amikor valami fontos történik egy publisher-en belül, értesíteni kell az összes subscriber-t.
6. A frissítési értesítési metódusok végrehajtása subscriber osztályokban.
7. A kliensnek kell létrehoznia az összes szükséges subscriber-t és regisztrálnia kell őket a megfelelő publisher-eknél.

Observer előnyök és hátrányok

- **Előnyök**
    - Open/Closed elv
    - Az objektumok közötti kapcsolatokat futás közben is létrehozhatjuk.
- **Hátrányok**
    - A subscriber-eket véletlenszerű sorrendben értesíti.

Template (Behavioral)

- Egy műveleten belül algoritmus vázat definiál és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
- **Probléma**
    - Készítünk egy olyan alkalmazást, amivel különböző dokumentumokból adatokat lehet kinyerni.
    - Egy idővel rájövünk, hogy például a PDF, CSV fájlok között viszonylag hasonló műveletek hajtódnak végre, így kód duplikáció keletkezhet.
- **Megoldás**
    - Magát az algoritmust bontsuk szét kisebb lépésekre, metódusokra.
    - Ezeket fogjuk meghívni a template method-ban.

Template használjuk, ha

- Algoritmust kisebb lépésekre szeretnénk bontani.
- Logikai hasonlóság esetén

Template implementálása

1. Kisebb részekre bontás
2. Absztrakt osztály létrehozása, ahol deklaráljuk a template method-ot.
3. Hívjuk meg az alosztályokat, a lépéseket a template method-ban.

Template előnyök és hátrányok

- **Előnyök**
    - Kód duplikáció elkerülhető vele, tehát a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg.
        - Leszármazott osztályban felül lehet definiálni.
- **Hátrányok**
    - Megsérthetjük a Liskov behelyettesítési elvet, ha egy alosztályon keresztül elnyomja az alapértelmezett lépés implementációját.
    - Template method-okat egy idő után nehéz karbantartani, ha sok kisebb lépést (metódusokat) tartalmaz.

IoC használata a gyakorlatban (MVVM Light / ASP.NET Core)

Gang-of-Four tervezési minták 3

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.
   a. Első híváskor az új objektum létrehozása és statikus mezőbe helyezése.
   b. A metódusnak minden további híváskor mindig ezt a példányt kell visszaadnia.
4. Az osztály konstruktora legyen privát.
   a. 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 nem 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.
    a. A konstruktornak az átadott objektumból az osztályban definiált összes mező értékét át kell másolnia az újonnan létrehozott példányba.
    b. 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.

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.
    a. Különböző metódusokat tartalmazhat az objektumok létrehozására.
5. Kliens kód használja a builder és a director objektumokat.
    a. Először a builder objektumot át kell adni a director-nak konstruktoron keresztül paraméterként.
    b. Innentől kezdve a director használja a builder-t.
6. Builder eredmény akkor születik a director-ból, ha minden elem ugyanazt az interfészt használja.
    a. 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.

Iterator

- Szekvenciális hozzáférést egy összetett (pl.: lista) objektum elemeihez anélkül, hogy annak belső reprezentációját felfedné.

- **Probléma**
    - Legyen bármilyen gyűjteményünk ezeket szeretnénk egy bejárható interfészen keresztül elérni.
- **Megoldás**
    - Adatszerkezeten implementáljuk az IEnumerable<T> és egy külső bejáró osztályon az IEnumerator<T> interfészt. Előírja ezeket a metódusokat:
        - **void Reset():** Gyűjtemény elejére visszaállás
        - **bool MoveNext():** Következő elemre lépés
        - **T Current:** Visszaadja az aktuális elemet.

Iterator használjuk, ha

- Úgy szeretnénk hozzáférni egy objektum tartalmazott objektumaihoz, hogy nem akarjuk felfedni a belső működését.
- Többféle hozzáférést szeretnénk biztosítani a tartalmazott objektumokhoz.
- Egy időben több, egymástól független hozzáférést szeretnénk a lista elemeihez.
- Különböző tartalmazott struktúrákhoz szeretnénk hozzáférni hasonló interfésszel.

Iterator előnyök és hátrányok

- **Előnyök**
    - Single Responsibility elv és Open/Closed elv
- **Hátrányok**
    - Iterator nem biztos, hogy olyan hatékony, mint egyes gyűjtemények elemeinek közvetlen végigjárása.

Chain of Responsibility (Behavioral pattern)

- Az üzenet vagy kérés küldőjét függetleníti a kezelő objektum(ok)tól.
- **Probléma**
    - Validálások szekvenciálisan hajtódnak végre.
    - Egy idő után bonyolulttá, átláthatatlanná válik a kód a sok validáció miatt.
        - Esetleges duplikációk is felmerülhetnek, mert lehet kell egy másik validáció miatt.
- **Megoldás**
    - Önálló objektumokká, handlerekké (handlers) alakítunk át.
    - Egy ellenőrzés egy metódus, paraméterként adunk át adatokat.
    - A handlereket láncba kell kapcsolni.
        - Minden összekapcsolt handler rendelkezik egy mezővel, ami a lánc következő handerre való hivatkozást tárolja.
    - A kérés feldolgozása mellett a handlerek továbbítják a kérést a láncban.
        - A kérés addig halad a láncban, amíg az összes handler fel nem tudja dolgozni azt.
    - A handler dönthet úgy, hogy nem továbbítja a kérést a láncba és ezzel leállítja a további feldolgozást.

Chain of Responsibility implementálása

1. Handler interfész deklarálása és a kérések kezelésére szolgáló metódus leírása.

2. A handlerekben található kódok kiküszöbölése érdekében érdemes létrehozni egy absztrakt alap kezelő osztályt, ami a handler interfészből származik.
    a. Kell egy mező, ami rámutat a következő handler-re.
    b. Ha a láncokat futásidőben módosítani akarjuk, akkor a referencia mező értékének megváltoztatására egy setter-t kell definiálni.
3. Egyenként hozzunk létre handler alosztályokat és a handler metódusokat valósítsuk meg.
    a. Minden handler-nek két döntést kell hoznia, amikor egy kérést fogad:
        i. Feldolgozza-e a kérést vagy továbbítja-e a kérést a láncban.
4. A kliens saját maga állíthat össze láncokat vagy más objektumoktól kaphat előre elkészített láncokat.
5. A kliens bármelyik handlert elindíthatja.
    a. A kérés addig halad végig a láncon, amíg valamelyik kezelő el nem utasítja a továbbküldést vagy amíg a lánc végére nem ér.
6. Kliensnek kész kell állnia ezekre:
    a. Egyetlen elem a láncban.
    b. Előfordulhat, hogy egyes kérések nem érik el a lánc végét.
    c. Kezeletlenül értek a lánc végére.

Chain of Responsibility használjuk, ha

- Több, mint egy objektum kezelhet le egy kérést és a kérést kezelő példány alapból nem ismert, automatikusan kell megállapítani, hogy melyik objektum lesz az.
- Egy kérést objektumok egy csoportjából egy objektumnak akarjuk címezni, a fogadó konkrét megnevezése nélkül.
- Egy kérést lekezelő objektumok csoportja dinamikusan jelölhető ki.

Chain of Responsibility előnyök és hátrányok

- **Előnyök**
    - Kérések kezelésének sorrendje szabályozható.
    - Single Responsibility elv, Open/Closed elv
- **Hátrány**
    - Néhány kérés kezeletlenül maradhat

Visitor (Behavioral pattern)

- Lehetővé teszi, hogy az algoritmusokat elválasszuk azoktól az objektumoktól, amiken azok működnek.
- **Probléma**
    - Hívó és hívott szétválasztása.
    - Hívó tudhat a hívottról, de fordítva tilos.
    - A hívott dönthessen róla, hogy lehet-e vele dolgozni éppen.
- **Megoldás**
    - Interfészeken át érjék el egymást.
    - Hívottnak legyen Accept() metódusa
    - Hívónak legyen Visit() metódusa
    - A hívott az Accept() metódusban döntést hoz és egyben meghívja a hívó Visit metódusát.

Visitor használjuk, ha

- Ha egy komplex objektumstruktúra (például object tree) összes elemén végre kell hajtani egy műveletet.
- Kiegészítő viselkedések üzleti logikájának (business logic) „tisztítására”.
- Ha egy behavior-nak csak az osztályhierarchia egyes osztályaiban van értelme, de más osztályokban nincs.

Visitor implementálása

1. Visitor interfész deklarálása „visiting” metódusokkal.
2. Element interfészének deklarálása.
    a. Ha egy meglévő elemosztály-hierarchiával dolgozunk, adjuk hozzá a hierarchia alaposztályához az absztrakt Accept() metódust.
    b. Ennek a metódusnak argumentumként egy látogató objektumot kell elfogadnia.
3. Accept() metódusok végrehajtása.
    a. Ezeknek a metódusoknak át kell irányítaniuk a hívást a bejövő visitor objektum metódusára, ami megfelel az aktuális elem osztályának.
4. A visitor-oknak ismerniük kell a Visit() metódusok paramétertípusaiként hivatkozott összes konkrét elemosztályát.
5. Minden olyan behavior-höz, ami nem valósítható meg az elemhierarchián belül, akkor hozzon létre egy új konkrét visitor osztályt és meg kell valósítani az összes visit metódust.
6. A kliensnek visitor objektumokat kell létrehoznia és azokat az accept (elfogadó) metódusokon keresztül átadni az elemekbe.

Visitor előnyök és hátrányok

- **Előnyök**
    - Open/Closed elv
    - Single Responsibility elv
- **Hátrányok**
    - Minden alkalommal frissíteni kell az összes visitor-t, amikor egy osztály hozzáadódik az elemhierarchiához vagy eltávolításra kerül belőle.
    - Előfordulnak, hogy a visitor-ok nem rendelkeznek a szükséges hozzáféréssel azon elemek privát mezőihez és metódusaihoz, amikkel dolgozniuk kell.

Gang-of-Four tervezési minták 4

Visitor (Behavioral pattern)

- Lehetővé teszi, hogy az algoritmusokat elválasszuk azoktól az objektumoktól, amiken azok működnek.
- **Probléma**
  - Hívó és hívott szétválasztása és a hívó tudhat a hívottról, de fordítva tilos.
  - A hívott dönthessen róla, hogy lehet-e vele dolgozni éppen.
- **Megoldás**
  - Interfészeken át érjék el egymást.
  - Hívottnak legyen Accept() metódusa és a hívónak legyen Visit() metódusa
  - A hívott az Accept() metódusban döntést hoz és egyben meghívja a hívó Visit metódusát.

Visitor használjuk, ha

- Ha egy komplex objektumstruktúra (object tree) összes elemén végre kell hajtani egy műveletet.
- Kiegészítő viselkedések üzleti logikájának (business logic) „tisztítására”.
- Ha egy behavior-nak csak az osztályhierarchia egyes osztályaiban van értelme.

Visitor implementálása

1. Visitor interfész deklarálása „visiting” metódusokkal.
2. Element interfészének deklarálása.
   a. Ha egy meglévő elemosztály-hierarchiával dolgozunk, adjuk hozzá a hierarchia alaposztályához az absztrakt Accept() metódust.
   b. Ennek a metódusnak argumentumként egy látogató objektumot kell elfogadnia.
3. Accept() metódusok végrehajtása.
   a. Ezeknek a metódusoknak át kell irányítaniuk a hívást a bejövő visitor objektum metódusára, ami megfelel az aktuális elem osztályának.
4. A visitor-oknak ismerniük kell a Visit() metódusok paramétertípusaiként hivatkozott összes konkrét elemosztályát.
5. Minden olyan behavior-höz, ami nem valósítható meg az elemhierarchián belül, akkor hozzon létre egy új konkrét visitor osztályt és meg kell valósítani az összes visit metódust.
6. A kliensnek visitor objektumokat kell létrehoznia és azokat az accept (elfogadó) metódusokon keresztül átadni az elemekbe.

Visitor előnyök és hátrányok

- **Előnyök**
  - Open/Closed elv, Single Responsibility elv
- **Hátrányok**
  - Minden alkalommal frissíteni kell az összes visitor-t, amikor egy osztály hozzáadódik az elemhierarchiához vagy eltávolításra kerül belőle.
  - Előfordulnak, hogy a visitor-ok nem rendelkeznek a szükséges hozzáféréssel azon elemek privát mezőihez és metódusaihoz, amikkel dolgozniuk kell.

Observer

- Hogyan tudják az objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól.
- **Az Observer az egyik leggyakrabban használt minta!**
- **Probléma**
    - Vevő szeretne vásárolni egy új terméket, de nem szeretne mindennap meglátogatni az üzletet, ahol lehet kapni.
    - Az üzlet pedig nem szeretné feleslegesen fogyasztani az erőforrásait abból a szempontból, hogy minden egyes új termék miatt küldözget emailt, mert ez csak spam lenne.
    - Tehát a vevő pazarolja a saját idejét vagy az üzlet az erőforrásait pazarolja.
- **Megoldás**
    - Kell egy subscriber, amivel feliratkozunk valamire és az értesít.
    - Feliratkozó osztályok megvalósítanak egy ISubscriber interfészt.
    - Írjon elő egy StateChange() vagy Update() metódust.
    - A subject kezelje a feliratkozókat Subscribe(), UnSubscribe()
    - Állapotváltozáskor hívja meg az összes feliratkozó StateChange() metódusát.
    - A feliratkozók tegyék meg a frissítési lépéseket.

Observer használjuk, ha

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

Observer implementálása

1. Business logic két részre bontása:
    a. Alapvető, más kódtól független funkcinalitás fog publisher-ként működni.
    b. A többi pedig subscriber osztályok halmaza lesz.
2. Subscriber interfész deklarálása és legalább egy frissítési metódust kell deklarálnia.
3. Publisher interfész deklarálása, subscriber-ben implementáljuk ezeket a metódusokat.
    a. A publisher-ek csak a subscriber-ekkel dolgozhatnak a subscriber interfészen keresztül.
4. Hozzunk létre egy absztrakt osztályt, ami közvetlenül a publisher interfészből származik.
    a. A publisher-ek kiterjesztik ezt az osztályt, örökölve a subscriber behavior-t.
5. Publisher osztályok létrehozása.
    a. Minden alkalommal, amikor valami fontos történik egy publisher-en belül, értesíteni kell az összes subscriber-t.
6. A frissítési értesítési metódusok végrehajtása subscriber osztályokban.
7. A kliensnek kell létrehoznia az összes szükséges subscriber-t és regisztrálnia kell őket a megfelelő publisher-eknél.

Observer előnyök és hátrányok

- **Előnyök**
  - Open/Closed elv
  - Az objektumok közötti kapcsolatokat futás közben is létrehozhatjuk.
- **Hátrányok**
  - A subscriber-eket véletlenszerű sorrendben értesíti.

Command (Behavioral pattern)

- Egy kérés objektumként való egységbezárása.
- Lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását.
- **Probléma**
  - Készítünk egy toolbar-t, amiben többféle gomb (button) található meg, amiknek mind különböző funkciójuk van.
  - Gombonként akkor külön alosztályokat kellene létrehozni.
  - Kód duplikáció is előfordulhat.
- **Megoldás**
  - Használjunk rétegzést, ezáltal külön választjuk a GUI-t és a business logic-ot.
  - A GUI felel a renderelésért, a business logic pedig végrehajtja a funkcionalitást.

Command használjuk, ha

- A strukturált programban callback függvényt használnánk, OO programban használjunk commandot helyette.
- Szeretnénk a kéréseket különböző időben kiszolgálni.
  - Ilyenkor várakozási sort használunk, a command-ban tároljuk a paramétereket, majd akár különböző folyamatokból/szálakból is feldolgozhatjuk őket.
- Visszavonás támogatására (eltároljuk az előző állapotot a command-ban).

Command implementálása

1. Command interfész deklarálása egy végrehajtási metódussal.
2. Interfészek implementálása az osztályokban.
   a. Minden osztálynak rendelkeznie kell a kérés paramétereinek tárolására szolgáló mezőkkel és a tényleges receiver objektumra való hivatkozással.
   b. A command konstruktorán keresztül kell inicializálni.
3. A sender osztályokhoz adjuk hozzá a parancsok tárolására szolgáló mezőket.
   a. A sender osztályok csak a command interfészen keresztül kommunikáljanak a commandjaikkal.
4. Commandok végrehajtása
5. A kliensnek ilyen sorrendben kell végrehajtania az objektumok inicializálást:
   a. Receiver-ek létrehozása
   b. Commandok létrehozása
   c. Senderek létrehozása

Command előnyök és hátrányok

- **Előnyök**
  - Elválasztja a parancsot kiadó objektumot attól, amelyik tudja, hogyan kell lekezelni.
  - Kiterjeszthetővé teszi a Command specializálásával a parancs kezelését.
  - Összetett parancsok támogatása.
  - Egy parancs több GUI elemhez is hozzárendelhető.
  - Könnyű hozzáadni új commandokat, mert ehhez egyetlen létező osztályt sem kell változtatni.
- **Hátrányok**
  - A kód bonyolultabbá válhat, mivel egy teljesen új réteget vezetünk be a sender és a receiver közé.

Mediator (Behavioral pattern) (miért szükséges és hogyan kell elkerülni a kétirányú függőségeket?)

- Olyan objektumot definiál, ami egységbe zárja, hogy objektumok egy csoportja hogyan éri el egymást.
- **Probléma**
  - Egyirányú függőség van két réteg között, ne legyen semmilyen irányú függés.
  - Egy közvetítő osztályok keresztül lehessen csak beszélgetni két osztálynak.
- **Megoldás**
  - Megoldja, hogy az egymással kommunikáló objektumoknak ne kelljen egymásra hivatkozást tárolniuk, így biztosítja az objektumok laza csatolását.

Mediator használjuk, ha

- Ha nehéz megváltoztatni néhány osztályt, mert azok szorosan kapcsolódnak egy csomó másik osztályhoz.
- Ha egy komponenst nem tudunk újrafelhasználni egy másik programban mert túlságosan függ más komponensektől.

Mediator implementálása

1. Keressük meg azokat az osztályokat, amiket függetlenebbé szeretnénk tenni.
2. Mediator interfész deklarálása.
   a. Döntő fontosságú, ha a komponens osztályokat különböző kontextusban szeretnénk újrafelhasználni.
3. Mediator osztály megvalósítása.
4. Mediator felelhet a komponensobjektumok létrehozásáért és megsemmisítéséért.
5. A komponenseknek a mediator objektumra való hivatkozást kell tárolniuk.
   a. A kapcsolat létrehozása általában a komponense konstruktorában történik, ahol a mediator objektumot adjuk át paraméterként.
6. Komponensek kódját módosítsuk úgy, hogy a többi komponense metódusai helyett a mediator értesítési metódusát hívják meg.

Mediator előnyök és hátrányok

- **Előnyök**
    - Single Responsibility elv
    - Open/Closed elv
    - Komponensek közötti kapcsolatok minimalizálása.
    - Könnyebben újrafelhasználható komponensek
- **Hátrányok**
    - Egy idő után a mediator-ből god object lehet.

Single Responsibility elv

- Minden osztály egy dologért legyen felelős és azt jól lássa el.
- **Ha nem követjük, akkor:**
    - Spagetti kód, átláthatatlanság
    - Nagy méretű objektumok
    - Mindenért felelős alkalmazások és szolgáltatások

Interpreter (Behavioral pattern)

- **Probléma**
    - Tetszőleges bemenetből tetszőleges kimenetet szeretnénk gyártani.
    - Például egy (3 + 4) – (2 + 2) stringből egy intet, aminek az értéke 3.
    - Értelmező programok írásának OOP reprezentációja az Interpreter minta.
- **Megoldás (Egyben implementálása is)**
    - Elkészítjük az írásjeleket reprezentáló osztályokat (Token)
    - Lexer elkészítése
    - Parser elkészítése

Interpreter használjuk, ha

- Ha a nyelv nyelvtana nem bonyolult.
- Ha a hatékonyság nem prioritás

Interpreter előnyök és hátrányok

- **Előnyök**
    - Könnyű megváltoztatni és bővíteni a nyelvtant.
    - A nyelvtan implementálása egyszerű.
- **Hátránya**
    - Nem hatékony

Memento (Behavioral pattern)

- Lehetővé teszi, hogy elmentse vagy visszaállítsa egy objektum előző állapotát anélkül, hogy felfedné az implementáció részeit.
- **Probléma**
    - Készítünk egy text editor alkalmazást, ahol különböző funkciókat implementálunk.
    - Biztosítani kell azt, hogy lehessen visszaállítani korábbi „állapotot/snapshotot”, ezt így menteni kell.
- **Megoldás**
    - Egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát.
        - Így az objektum állapota később visszaállítható.

Memento implementálása

1. Originator osztály létrehozása
2. Memento osztály létrehozása, ahol hozzuk létre ugyanazokat a field-eket, amik az Originator osztályban vannak.
3. A Memento osztálynak nem szabad változtathatónak lennie (immutable), így csak konstruktoron keresztül kaphat értékeket.
4. Metódus hozzáadása, ami visszaadja a korábbi állapotot Originator osztályba, ami Memento objektumot várhat paraméterként.
5. Caretaker gondoskodik a tárolásról, ami tárolja az állapotokat, eldönti, hogy mikor kell visszaállítani.

Memento használjuk, ha

- Egy objektum (rész)állapotát később vissza kell állítani és egy közvetlen interfész az objektum állapotához használná az implementációs részleteket, vagyis megsértené az objektum egységbezárását.

Memento előnyök és hátrányok

- **Előnyök**
    - Megőrzi az egységbezárás határait.
- **Hátrányok**
    - Erőforrásigényes
    - Nem mindig jósolható meg a Caretaker által lefoglalt hely

State (Behavioral pattern)

- Lehetővé teszi egy objektum viselkedésének megváltozását, amikor megváltozik az állapota.
- **Probléma**
  - Túl nagy switch-case szerkezet, sok állapot = sok ellenőrzés
- **Megoldás**
  - Kontextust hozunk létre, ami az egyik állapotra hivatkozást tárol.

State implementálása

1. Hozzunk létre egy osztályt, ami lesz a kontextus (context).
2. State interfész létrehozása, hozzuk létre az állapot-specifikus viselkedést tartalmazó metódusokat.
3. Minden aktuális állapothoz hozzunk létre egy osztályt, ami implementálja a State interfészt.
4. Context osztályban deklaráljunk egy referencia mezőt a State interfész típusához, aminek legyen egy publikus setter-je, amivel felül lehet írni az értékét.
5. Megfelelő állapotfeltételhez hívjuk meg a megfelelő metódust.
6. Kontextus állapotának megváltoztatásához létre kell hozni egy példányt az egyik state osztályból és azt adjuk át a kontextusnak.

State használjuk, ha

- Az objektum viselkedése függ az állapotától és a viselkedését az aktuális állapotnak megfelelően futás közben meg kell változtatnia.
- A műveleteknek nagy feltételes ágai vannak, amik az objektum állapotától függenek.

State előnyök és hátrányok

- **Előnyök**
  - Egységbezárja az állapotfüggő viselkedést, így könnyű az új állapotok bevezetése.
  - Áttekinthetőbb kód, nincs nagy switch-case szerkezet
  - A State objektumot meg lehet osztani.
- **Hátrányok**
  - Nő az osztályok száma, így csak indokolt esetben használjuk.

Composite (Structural pattern)

- Másnéven **Object Tree**
- **Probléma**
    - Nehezen tudunk az objektumainkból hierarchikus rendszert építeni.
    - Például részlegek és dolgozók korrekt ábrázolása.
    - Egy részfa vagy akár egy levélelem is ugyanazt a szolgáltatáskészletet nyújtsa.
- **Megoldás**
    - Fa szerkezet építése
    - Egy csomópontnak tetszőleges mennyiségű gyermekeleme legyen.
    - A csomópontnak és levél elemek is ugyanazt az interfészt valósítsák meg.
    - Lehessen rekurzívan bejárni.

Composite implementálása

1. Alkalmazás alapvető modellje fa struktúraként ábrázolható kell legyen.
2. Komponens interfész implementálása
3. Levélosztály létrehozása az egyszerű elemek ábrázolására.
4. Osztály létrehozása az összetett elemek ábrázolásához.
    a. Tömböt létre kell hozni, amiben az alelemekre való hivatkozásokat tárolja.
    b. Tömbnek képesnek kell lennie a levelek, konténerek tárolására is, ezért a komponens interfész típusával kell deklarálni.
5. Metódusok deklarálása, amivel hozzáadhatunk vagy törölhetünk gyermekelemeket.

Composite használjuk, ha

- Objektumok rész-egész viszonyát szeretnénk kezelni.
- A kliensek számára el akarjuk rejteni, hogy egy objektum egyedi objektum vagy kompozit objektum.
    - Bizonyos szempontból egységesen szeretnénk kezelni őket.

Composite előnyök és hátrányok

- **Előnyök**
    - Összetetteb fa struktúrával is dolgozhatunk.
    - Open/Closed elv
- **Hátrányok**
    - Nehéz lehet közös interfészt biztosítani, mivel a funkcionalitások eltérhetnek.

Gang-of-Four tervezési minták 5

Open/Closed elv

- Egy osztály legyen nyitott a bővítésre és a zárt módosításra, vagyis nem írhatunk bele, de származtathatunk tőle.
- **Ha nem követjük, akkor:**
  - Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetőek.
  - Leszármazott megírásakor módosítani kell az ősosztályt, ami tilos.
  - Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában.

Liskov behelyettesíthetőség elve

- Ősosztály helyett utódpéldány legyen mindig használható.
- Compiler supported, hiszen OOP elv (polimorfizmus)
- Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.

OOP virtuális metódusok

Késői kötés

- A metódus meghívásakor a referencia által hivatkozott objektum valódi típusának megfelelő osztály metódusa hívódik meg.
- A fordító nem tudhatja, hogy a meghívás pillanatában milyen típusú objektumra fog hivatkozni a referencia, így olyan kódot generál, ami futás közben dönti el, hogy melyik metódust hívja meg, ez a dinamikus kötés.
- **Előnyei:** Rugalmasság, polimorfizmus lehetősége
- **Hátrányai:** Teljesítményveszteség, tárigény, bonyolult

Virtuális metódus

- Minden meghívásakor késői kötést fog használni (**virtual**).
- Leszármazottakban felül lehet definiálni (**override**), ilyenkor a késői kötésnek megfelelően fog a megfelelő metódus lefutni.

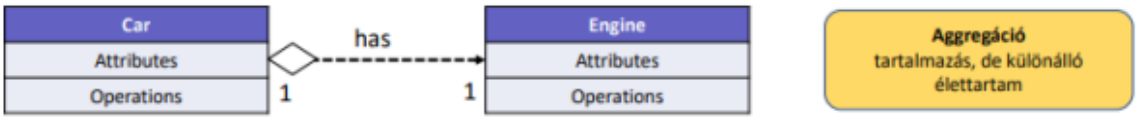
VMT – Virtuális Metódus Tábla

- Egy objektum VMT táblázata tartalmazza a dinamikusan kötött metódusok címeit.
- Egy metódus meghívásakor a VMT alapján dönthető el futásidőben, hogy melyik megvalósításnak kell lefutnia.
- Azonos osztályba tartozó objektumoknál ez mindig azonos, ezt megszokták osztani és minden objektum csak egy mutatót tárol erre a táblázatra.

Kompozíció

Aggregáció

- Az asszociáció speciális esete, tartalmazási kapcsolat.
- A tartalmazó osztály példányai magukba foglalják a tartalmazott osztály egy vagy több példányát, ez a rész-egész kapcsolat.
- A tartalmazó és a tartalmazott osztály egymástól függetlenül létezhetnek.
- A tartalmazás lehet, erős illetve gyenge.
- **Erős aggregáció:** A részek élettartalma szigorúan megegyezik az egészével, ez a kompocízió.
- **Gyenge aggregáció:** Egyszerű/általános aggregáció.



Kompozíció

- Másnéven **erős aggregáció**, tehát szigorú tartalmazási kapcsolat.
- Egy rész objektum csak egy egészhez tartozhat.
- **A tartalmazó és a tartalmazott életciklusa közös**: Például van egy User objektum, aminek van egy Address tulajdonsága. Ha a User objektum megszűnik, akkor megszűnik az Address is, de nem létezhet User objektum Address nélkül. Ezért közös az életciklusuk.

Strategy (Behavioral pattern)

- Algoritmusok egy csoportján belül az egyes algoritmusok egységbezárása és egymással kicserélhetővé tétele.
- A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetőek.
- **Probléma**
  - Készítünk egy navigációs alkalmazást, ahol időről időre új funkciókat szeretnénk lefejleszteni.
  - Útválasztási algoritmusokat akarunk fejleszteni autókhoz, sétához, stb.
  - Átláthatatlan lett a kód, mert amikor hozzáadunk mindig egy új útválasztási algoritmust, megduplázódott a kód.
- **Megoldás**
  - Adott algoritmust szervezzük ki egy külön osztályba, ezt nevezzük **stratégiának.**

Strategy használjuk, ha

- Egy objektumon belül egy algoritmus különböző változatait szeretnénk használni és képesnek kell lennie a változásra az egyik algoritmusról a másikra futás (runtime) közben.
- Sok hasonló osztály van, amik csak abban különböznek egymástól, hogy hogyan hajtanak végre bizonyos viselkedést.
- Business Logic elkülönítése

Strategy implementálása

1. Context osztályban azonosítsunk be egy olyan algoritmust, ami hajlamos a gyakori változásokra.
2. Strategy interfész deklarálása
3. Egyenként implementáljuk a Strategy interfészt a megfelelő algoritmussal a saját osztályába.
4. Context osztályban legyen egy field, ami tárolja a strategy objektum referenciáját tárolja.
    a. Setter-t is állítsunk neki

Strategy előnyök és hátrányok

- **Előnyök**
    - Objektumon belül algoritmust tudunk cserélni futásidőben. (runtime)
    - Az algoritmus implementálását izolálhatjuk az azt használó kódtól.
    - Öröklődést kicserélhetjük kompozícióval.
    - Open/Closed elv
- **Hátrányok**
    - Ha csak pár algoritmus van és azok is ritkán változnak, akkor nem érdemes túlbonyolítani új algoritmusokkal, osztályokkal és interfészekkel, amik a mintával együtt járnak.

Template (Behavioral)

- Egy műveleten belül algoritmus vázat definiál és ennek néhány lépésének implementálását a leszármazott osztályra bízza.
- **Probléma**
    - Készítünk egy olyan alkalmazást, amivel különböző dokumentumokból adatokat lehet kinyerni.
    - Egy idővel rájövünk, hogy például a PDF, CSV fájlok között viszonylag hasonló műveletek hajtódnak végre, így kód duplikáció keletkezhet.
- **Megoldás**
    - Magát az algoritmust bontsuk szét kisebb lépésekre, metódusokra.
    - Ezeket fogjuk meghívni a template method-ban.

Template használjuk, ha

- Algoritmust kisebb lépésekre szeretnénk bontani.
- Logikai hasonlóság esetén

Template implementálása

1. Kisebb részekre bontás
2. Absztrakt osztály létrehozása, ahol deklaráljuk a template method-ot.
3. Hívjuk meg az alosztályokat, a lépéseket a template method-ban.

Template előnyök és hátrányok

- **Előnyök**
  - Kód duplikáció elkerülhető vele, tehát a hierarchiában a közös kódrészeket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg.
    - Leszármazott osztályban felül lehet definiálni.
- **Hátrányok**
  - Megsérthetjük a Liskov behelyettesítési elvet, ha egy alosztályon keresztül elnyomja az alapértelmezett lépés implementációját.
  - Template method-okat egy idő után nehéz karbantartani, ha sok kisebb lépést (metódusokat) tartalmaz.

Adapter (Structural pattern)

- Egy osztály interfészét olyan interfésszé konvertálja, amit a kliens vár.
- Lehetővé teszi olyan osztályok együttműködését, amik egyébként inkompatibilis interfészeik miatt nem tudnának együttműködni.
- **Probléma**
  - Össze szeretnénk kötni két rendszert, amik nem kompatibilisek.
    - Például van egy alkalmazás, ami XML formátummal működik és szeretnénk használni egy másik csomagot, ami csak JSON formátummal működik.
- **Megoldás**
  - *(Valós példa: Adapter kábelek: VGA -> HDMI és vissza)*
  - Készítünk egy adapter-t, ami elrejti magát a konverziót.

Adapter használjuk, ha

- Egy olyan osztályt szeretnénk használni, aminek interfésze nem megfelelő Adapter.
- Egy újrafelhasználható osztályt szeretnénk készíteni, ami együttműködik előre nem látható vagy független szerkezetű osztályokkal. (pluggable adapters)

Adapter implementálása

1. Adapter osztály elkészítése
2. Az adapter osztályban adjunk hozzá egy field-et, ami referenciaként rámutat a service objektumra.
3. Kliens interfész metódusainak implementálása az adapter osztályban.
4. Hajtsuk végre magát a konverziót az adapter segítségével a két nem kompatibilis interfész között.

Adapter előnyök és hátrányok

- **Előnyök**
  - Single Responsibility elv
  - Open/Closed elv
- **Hátrányok**
  - Komplexitás növekedhet minden egyes új osztálynál és interfésznél.

Bridge (Structural pattern)

- Különválasztja az absztrakciót (interfészt) az implementációtól, hogy egymástól függetlenül lehessen őket változtatni.
- **Probléma**
    - Egy osztály két jellemzőtől is függ
    - Például alakzatok, szín és forma
- **Megoldás**
    - Szét kell bontani az osztályt
    - A forma osztály várja interfészen keresztül a szín osztályt
    - Kompozícióval lehessen összeépíteni őket

Bridge használjuk, ha

- Egy osztályt több ortogonális (független) dimenzióban kell bővíteni.
- Futás közben implementációt szeretnénk váltani

Bridge implementálása

1. Bridge interfész létrehozása.
2. Bridge osztály létrehozása, ami implementálja a Bridge interfészt.
3. Abstract osztály létrehozása
4. Konkrét osztály létrehozása, ami implementálja az Abstract osztályt

Bridge előnyei és hátrányai

- **Előnyök**
    - Absztrakció és az implementáció különválasztása
    - Az implementáció dinamikusan, akár futási időben is megváltoztatható
    - Az implementációs részletek a klienstől teljesen elrejthetők
    - Az implementációs hierarchia külön lefordított komponensbe tehető, így ha ez ritkán változik, nagy projekteknél nagymértékben gyorsítható a fordítás ideje
    - Ugyanaz az implementációs objektum több helyen is felhasználható
- **Hátrányok**
    - Bonyolulttá válhat a kód egy idő után

Dependency Injection elve és megvalósítása

- Lazán csatoltság kiterjeszthetővé teszi a kódot, a kiterjeszthetőség pedig karbantarthatóvá.
- **Probléma**
    - A kód függjön absztrakciótól, ne konkrét implementációtól.
- **Megoldás**
    - Az interfészt várjuk paraméterként a konstruktorban.
    - Setter injektálás, amikor az objektumokat setter metódusok segítségével injektáljuk.

Gang-of-Four tervezési minták 6

Struktúrális minták

Facade (Structural pattern) („Homlokzat”)

- **<u>Egységes interfészt definiál egy alrendszer interfészeinek halmazához.</u>**
- **Probléma**
  - Kód széleskörű objektumokkal rendelkezik, amik egy library-hez vagy keretrendszerhez tartozik.
  - Normális esetben az összes objektumot inicializálni kellene, nyomon követni a függőségeket, a metódusokat a megfelelő sorrendben végrehajtani és így tovább.
  - Ennek eredményeként az osztályok üzleti logikája szorosan összekapcsolódik a harmadik féltől származó osztályok megvalósítási részleteivel, ami megnehezíti a megértést és a karbantartást.
- **Megoldás**
  - Csak a tényleges funkciókat tartalmazza.
  - Praktikus, ha integrálni kell az alkalmazást kell egy library-vel, ami sok funkcióval rendelkezik, de csak egy kis részére van szükség.
    - **Példa:** Egy alkalmazás, ami rövid videókat tölt fel egy platformra, ami egy összetett videókonvertáló könyvtárat használ.
    - Tehát azon az osztályon belül eléri azt a metódust, amivel lehet konvertálni és azt hozzácsatoljuk a konvertáló library-vel, akkor megvan az első facade.

Facade használati esetek

- Akkor használjuk, ha egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé.
- Akkor használjuk, ha számos függőség van a kliens és az alrendszerek osztályai között.
- Rétegeléskor

Facade implementációja

1. Nézzük meg, hogy lehet-e egyszerűbb interfészt biztosítani, mint amit egy meglévő alrendszer biztosít.
2. Interfész implementálása az új facade osztályban.
3. A facade-nek át kell irányítania a kódból érkező hívásokat az alrendszer megfelelő objektumaihoz.
4. Innentől kezdve a kódban csak a facade-en keresztül kommunikáljon az alrendszer.
   a. Mostantól a kód védve van az alrendszer kódjának bármilyen változásától.
   b. Ha egy alrendszer új verzióra frissül, csak a facade kódot kell módosítani.

Facade előnye és hátránya

- **Előny**
  - Elszigetelhető a kód az alrendszer komplexitásától.
- **Hátrány**
  - „god object” lehet belőle

Proxy (Structural pattern)

- **Objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést.**
- **Probléma:** Miért akarjuk ellenőrizni az objektumhoz való hozzáférést?
    - Van egy hatalmas objektum, ami rengeteg rendszererőforrást fogyaszt és időnként szükség van rá, de nem mindig.
- **Lusta megoldás**
    - Csak akkor hozzuk létre az objektumot, amikor tényleg szükség van rá.
    - Végre kellene hajtani néhány késleltetett inicializálási kódot, de ez kód duplikációt okozna.
- **Megoldás**
    - Hozzunk létre egy új proxy osztályt, aminek interfésze megegyezik az eredeti service objektummal.
    - Ezután frissíti az alkalmazást, hogy átadja a proxy objektumot az eredeti objektum összes kliensének.
    - A klienstől érkező kérés fogadásakor a proxy létrehoz egy valódi service objektumot és mindent átad neki.
- **Haszna**
    - Ha valamit az osztály alapvető logikája előtt vagy után kell végrehajtani, a proxy lehetővé teszi, hogy ezt az osztály megváltoztatása nélkül tegye.
    - Mivel a proxy ugyanazt az interfészt valósítja meg, mint az eredeti osztály, átadható bármely olyan kliensek, ami valódi szolgáltatásobjektumot vár.

Proxy struktúra

- **Subject:** Közös interfészt biztosít a Subject és a Proxy számára (így tud a minta működni).
- **Realsubject:** A valódi objektum, amit a proxy elrejt.
- **Proxy:** Helyettesítő objektum, ami tartalmaz egy referenciát a tényleges objektumra, hogy el tudja azt érni. Szabályozza a hozzáférést a tényleges objektumhoz, feladata lehet a tényleges objektum létrehozása, törlése.

Proxy típusok

- **Távoli Proxy:** Távoli objektumok lokális megjelenítése „átlátszó” módon, tehát a kliens nem is érzékeli, hogy a tényleges objektum egy másik címtartományban vagy egy másik gépen van.
- **Virtuális Proxy:** Nagy erőforrás igényű objektumok szerinti létrehozása, például egy kép.
- **Védelmi Proxy:** A hozzáférést szabályozza különböző jogoknál.
- **Smart Pointer:** Egy pointer egységbezárása, hogy bizonyos esetekben automatikus műveleteket hajtson végre, például lockolás.

Proxy implementációja

1. Service interfész létrehozása vagy a proxy a service osztály alosztálya lesz és így örökli a service interfészét.
2. Proxy osztály létrehozása és egy field-et deklarálni kell, hogy lehessen hivatkozni a service-re.
3. Proxy metódusok implementálása.
4. Meg kell fontolni egy olyan létrehozási módszer bevezetését, ami eldönti, hogy a kliens proxy vagy valódi service-t kap-e. (Ez lehet egy statikus vagy factory metódus is.)
5. Service objektum inicializálása.

Proxy előnyei és hátrányai

- **Előnyök**
  - A service objektumot a kliensek tudta nélkül is lehet vezérelni.
  - Akkor is működik a proxy, ha a service objektum nem áll készen vagy nem elérhető.
  - Open/Closed elv alapján működik, tehát a service vagy a kliensek módosítása nélkül új proxy-kat lehet bevezetni.
- **Hátrányai**
  - Bonyolult kód sok új osztálynál.
  - A service válasza késhet.

Decorator (Structural pattern)

- **Objektumok funkciójának dinamikus kiterjesztése, vagyis rugalmas alternatívája a leszármaztatásnak.**
- **Probléma**
  - Van egy notification library, amit más program arra használnak, hogy fontos eseményekről küldjön értesítést.
  - Használatkor kiderül, hogy csak email-eket lehet vele küldeni, és a programban pedig SMS-eket szeretne küldeni és így tovább.
  - Így alosztályokat hozunk létre, amik több értesítési módszert kombinálnak egy osztályon belül, de ez azért **nem jó**, mert a könyvtári és a kliens kódot is megnöveli nagy mértékben.
- **Megoldás**
  - Decorator-öket kell csinálni a különböző metódusokból, például az értesítő módszereknél, csinálunk SMS, Facebook, stb decorator-öket.
  - A decorator-ök ugyanazokat az interfészeket használják.
  - Példaként ha fázom, akkor felveszem egy pulóvert és ha még mindig fázom, akkor egy kabátot is felveszek.

Decorator használati esetek

- Akkor használjuk, ha dinamikusan szeretnénk funkcionalitást/viselkedést hozzárendelni az egyes objektumokhoz.
- Akkor használjuk, ha a funkcionalitást a kliens számára átlátszó módon szeretnénk az objektumhoz rendelni.
- Akkor használjuk, amikor a származtatás nem praktikus.

Decorator előnyei és hátrányai

- **Előnyei**
  - Sokkal rugalmasabb, mint a statikus öröklődés.
  - Több testreszabható osztály határozza meg a tulajdonságokat.
- **Hátrányai**
  - Bonyolultabb, mint az egyszerű öröklés, mert több osztály szerepel.
  - A decorator és a dekorált komponens interfésze azonos, de maga az osztály nem ugyanaz.

Flyweight (Structural pattern) trükkök

- Nincs konkrét megoldás, sok trükköt biztosít a Flyweight minta.

On-the-fly property-k

- A memóriában nem foglalnak helyet ezek a property-k.
- Amikor az adott property-t lekérjük, akkor lazy loading elven akkor hajtódik végre, amikor szükség van rá.
- Amikor a főprogram elkéri az adott property-t, akkor hajtódik végre a „levegőben”, emiatt nevezzük on-the-fly property-nek.
- El kell dönteni, hogy mikor akarjuk használni, mert például ha rengeteg adat van és például azokon akarunk átlagolni, akkor az sokáig is eltarthat.
- Ha nem használjuk, akkor pedig használjunk külön szálakat, aszinkron metódusokat például.

Objektumok közös részeinek eltárolása egyszer

- Példány szintjén is megnézhetjük az adott tulajdonságot.
- Felesleges tárolást lehet vele kiváltani, mert olyan jellemzőket teszünk bele, amiket nem szeretnénk módosítani.
- Mivel ez egy megosztott objektum és ha átírunk valamit, akkor az összes többi példányra kihatással van.
- Így érdemes védeni az írás ellen, tehát olvashatóként kell definiálni.

Újrahasznosított objektumok

- Lényege, hogy ne hozzunk létre újabb objektumot például egy törlés után, hanem használjuk fel újra a már meglévőt.
- Memóriát és CPU időt is megtakaríthatunk vele, mert mindig ugyanazt az objektumot használjuk fel.

Flyweight a .NET osztályokban (String, Type)

- **String**-ek .NET-ben immutable-ek, vagyis nem lehet létrehozás után módosítani.
    - Gyorsítótárba helyezi újrafelhasználás céljából.
    - Tehát megnézi, hogy van-e már egy ugyanilyen értékű létező String a String pool-ban, ha van, akkor nem jön létre új String, hanem a meglévő String-re való hivatkozás kerül vissza.
- A **Type** osztály egy objektum típusát reprezentálja és minden típusnak egyedi identitása van egy AppDomain-en belül.
    - Típusokat metaadatokból tölti be és a típus metaadatai az újrafelhasználás miatt gyorsítótárba kerülnek.
    - Ezek a metaadatok tartalmazzák a típus nevére, névterére, attribútuamira és member-ekre vonatkozó információkat.
    - Tehát a gyorsítótárazott metaadatokat adja vissza, ahelyett, hogy a metaadatokat újratöltené a lemeről.

Domain-Driven-Design filozófia

Domain-Driven-Design

Lényege

- A rétegzés ne attól függjön, hogy MVC vagy API vagy bármilyen a UI elérési technika.
    - Attól függjön, hogy mit akarok csinálni az adattal.
- Nem az adatbázissal kezdjük a modellezést, hanem a funkciókkal.

Bounded Context

- **Bounded context-eket hozunk létre:**
    - Domain model lesz belőlük
    - Jelentése, hogy egy user tábla szerepelhet a szállítás domain model-ben és a számlázás model-ben is.
    - DRY-elveknek ellentmond, de csak látszólag mert a Data Mapper / ORM majd valójában ugyanarra az 1db táblára mappeli le.
- **Hibalehetőségek:**
    - Bloated domain objects (túl sok felelőség)
    - Anemic domain objects (túl kevés felelőség)

Hexagon/Onion architektúrától a flexibilis rétegzésig

Onion architektúra

Hexagon architektúra

DDD megközelítés az írás-olvasás szétválasztásához

- Nagy rendszereknél általában SOK olvasási művelet és KEVÉS írási művelet történik.
- Írási műveletek
    - Tipikusan egy bounded context-be akarunk írni.
    - Kell minden alrendszer hozzá
- Olvasási műveletek
    - Dashboard és Reports funkcionalitás nagyon gyakori.
    - Általában több bounded context-ből kell összeszedni az adatokat.
    - Egy csomó alrendszer kikerülhető akár.

CQRS optimalizáció a lekérdezés (query) és az utasítás (command) oldalon

Query oldal – Olvasási műveletek

- Egyedi kérések problémája
    - Adatbázisok optimalizálhatóak kérésekre.
    - Gyors keresésre optimalizált DB: ElasticSearch

Command oldal – Írási műveletek

- Hibára futás ritka
- Szinkron hibajelzés feleslegesen lassít
- Aszinkron hibajelzés
    - Sikeres foglalás
    - Ha baj van, akkor email küldése, hogy hiba történt.
    - Aszinkron reagáló mechanizmusok

Event sourcing

- Probléma, hogy gyorsabban jön az input, minthogy fel tudnánk dolgozni.
- Például egy szenzor akarna 1mp-enként adatot küldeni, de a szerver annyira túlterhelt, hogy 3 mp múlva jön meg a HTTP response.
    - **Feltorlódnak a kérések és használhatatlan lesz a rendszer.**
- **Megoldás:**
    - Várósorba mentés, vagyis Event Store
    - Technika
        - Redis
        - RabbitMQ
        - MQTT
        - HiveMQ
    - Ezek az adatbázisok arra vannak optimalizálva, hogy villámgyorsan képesek legyenek elmenteni kéréseket, nagyságrendekkel gyorsabban, mint egy relációs adatbázis.

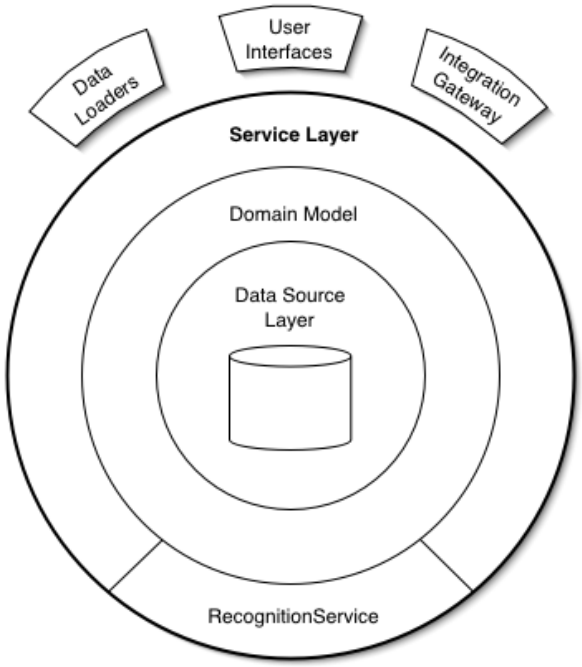
Event Sourcing + CQRS + DDD

- **Előnyei**
    - Nagy teljesítmény
    - Egyszerűbb a rendszerek összeépítése
    - Könnyű hibakeresés, tesztelés
    - Event Store-ból extra üzleti adat is kinyerhető.
- **Hátrányai**
    - Reporting bonyolult
    - Nagyobb tárigény
    - Hibás kérések visszajelzése nem azonnali

Mikroszervíz architektúra

Service réteg (domain/infrastructure logika szétválasztása)

- Domain model egységbezárja a Domain Entityket és azok üzleti műveleteit.
- Service-Oriented Architektúra (**Microservices**)
- **Feladata:**
    - Hívások fogadása, továbbítása a Domain Logic felé.
    - Tranzakciókezelés és a lock is.
- Alsóbb rétegekben megjelenik ettől függetlenül az adatbázis szintű tranzakciókezelés is.
- **3 részből áll**
    1. Domain/functional model
    2. REST endpointok sorozata
    3. A domain objektumok tárolására szolgáló eszköz vagy perzisztencia réteg.



SOA-koncepció

- A SOA amiben nem kód, hanem futó kód alapú elemekből állítható össze az alkalmazás.
    - Emiatt nem kell foglalkozni az adott funkcionalitás futtató környezetével, mert ezt egy megfelelő szolgáltatás végzi el és mi csak azt használjuk.

Szolgáltatások felépítése és összekapcsolása

Belső működés

- Hagyományos rétegezéssel épül fel.
- Saját adatbázissal rendelkezik.
- **Kommunikáció**
    - REST API vagy Message bus protokollok

Külső elérés

- Általában REST API-n keresztül.
- UI is egy mikroszolgáltatás, ami megjelenítésért felelős.

Üzenetszóró protokollok és szolgáltatások (AMQP/MQTT).

AMQP – Advanced Message Queueing Protocol

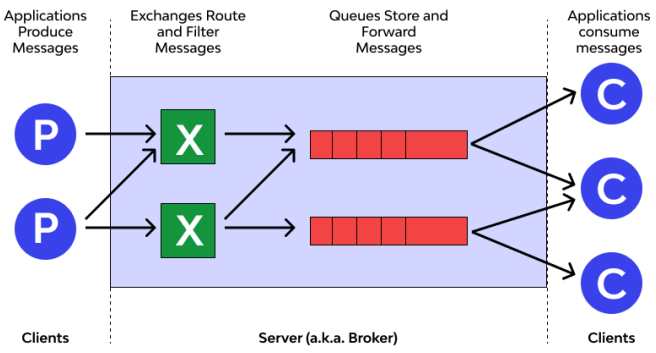
- Általános nyílt protokoll
- Általában PC/WEB
- Alkalmazási rétegen működik kliens és „brókerek” között.
- **A bróker vagy szerver döntő szerpet játszik az AMQP protokoll engedélyezésében.**
    - Felelős a kapcsolatépítésért, ami biztosítja a jobb adatátirányítást és a sorba állítást a kliens oldalon.
- Az üzenetek visszaigazolását a consumer/fogyasztó végzi.

AMQP komponensek

- **Exchanges:** Gondoskodik az üzenetek lekéréséről és elhelyezése a queue-ban.
    - **Kategóriák:** Fanout, Headers, Topic és Direct
- **Channel:** Az AMQP peerek közötti multiplexelt virtuális kapcsolatra utal, ami egy meglévő kapcsolaton belül épül fel.
- **Message queue:** Azonosított entitás, ami segít összekapcsolni az üzeneteket a forrásaikkal vagy a kiindulási pontjukkal.
- **Binding:** Az üzenetek küldését és kézbesítését kezeli.
- **Virtual Hosztok:** Lehetővé teszi, hogy különböző alkalmazások ugyanazt a broker-t használják anélkül, hogy zavarnák egymást.

AMQP Exchange működése

- **Direct exchange:** Összehasonlítja a routing kulcsot a hozzá kötött queue-k routing kulcsaival és az üzenetet bármelyik queue-nak kézbesíti, aminek routing kulcsa megegyezik az üzenet routing kulcsával.
- **Fanout exchange:** Az üzeneteket az összes olyan queue-ra kézbesítik, ami exchange-hez van kötve. A routing kulcsot figyelmen kívül hagyjuk és minden üzenetet az összes kötött queue-ra továbbítunk.
- **Topic exchange:** Az üzenetek egy vagy több, pontokkal elválasztott szóból álló routing kulcs alapján kerülnek a queue-kra. Az exchange a routing kulcsot összehasonlítja a hozzá kötött queue-k routing kulcsaival és a routing kulcsnak való megfeleléshez wildcard-okat használ.



AMQP API fejlesztés

- Direct üzenetek küldése
- Cache üzenetek sorba állítása a trigger-alapú küldéshez.
- Információkat továbbíthat vagy exchange-eket köthet a kijelölt queue-okhoz.
- Kapcsolatot teremt az exchange-k között a hatékony kommunikáció biztosítása érdekében.
- Automatikus vagy manuális visszaigazolást küldhet.

Mire használhatjuk az AMQP-t API-kban?

- Pénz feltöltése a digitális pénztárcákhoz
- Hitel-vagy betéti kártyás tranzakció kiskereskedelmi üzletekben
- Kommunikációs rendszerekben

AMQP példa folyamat

1. Pénz feltöltése digitális pénztárcához, valamilyen szolgáltatás segítségével, mint például Paypal-al.
2. Következő lépésként autentikációs lépés, majd ezután kerül feltöltésre ténylegesen a pénz a tárcához.
3. Előfordulhat, hogy a felhasználó meggondolta magát és nem szeretne feltölteni pénzt, így rendelkeznie kell egy törlési szolgáltatással.
4. Az AMQP üzenetváltás aszinkron módon történik.
    a. Ez azt jelenti, hogy a tranzakciók kritikus fontosságú események és nem lehet bennük inkonzisztencia.
    b. Lehet még egy esemény, ami nyomon követi az üzenetek kimenetelét, hogy a következő eseményt időben lehessen elindítani.

MQTT – Message Queue Telemetry Transport

- Egy ISO szabvány (ISO/IEC PRF 20922), publish-subscribe alapú üzenetküldő protokoll.
- Alacsony a sávszélesség igénye.
- Szükséges hozzá egy szerver, vagyis broker.

Broker feladata

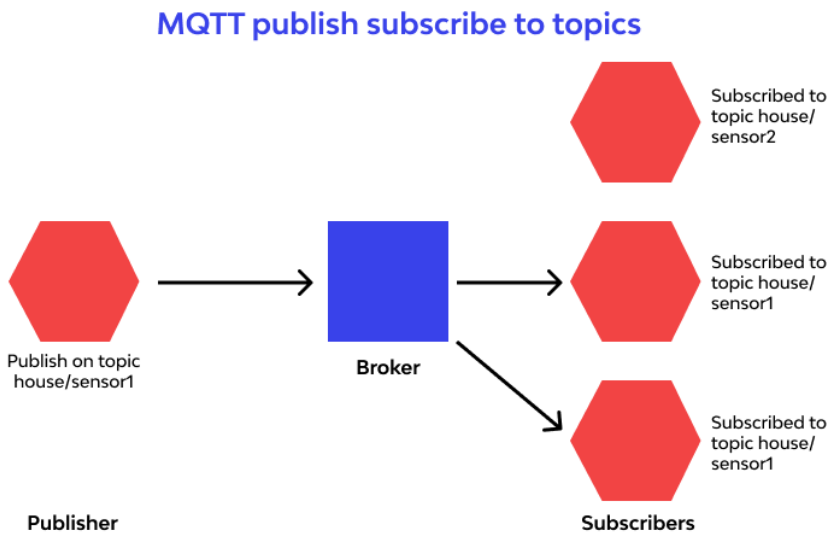
- Broker feladata a beérkező üzenetek továbbküldése a klienseknek, téma alapján.
    - Azaz a kliensek feliratkoznak különböző témákra, majd a broker a témákba érkezett üzeneteket továbbítja a feliratkozott klienseknek.

MQTT protokoll jellemzői

- **Az MQTT-t tartják a legmegfelelőbbnek az IoT számára, mivel rendelkezik bizonyos tulajdonságokkal:**
    - Könnyű használni és azonnali használatra elkészített broker-t és klienst kínál.
    - Csökkenti az alkalmazásfejlesztési időt.
    - Megbízható kapcsolatot kínál, mert az MQTT csökkenti a csatlakozási problémát egy QoS funkcióval, ami sorba állítja az üzeneteket és elmenti őket az MQTT broker-nél és megváratja őket, amíg a célzott eszköz készen áll az elfogadásra.
        - Ez csökkenti az üzenetek rossz elhelyezésének esélyét, így az üzenet biztosan eljut a célállomásra.
    - Skálázható üzenetek

MQTT működése

- Az MQTT modellben a kommunikáció közvetve, a PUSH/SUBSCRIBE topológián keresztül történik.
- A kliens, például az MQTT Explorer, csatlakozik a kapcsolódó publisher-hez és továbbítja az üzenetet.
- Ezt a megosztott adatot leválasztják a kliensről és továbbítják a következő szakaszba.
- A folyamat során a broker megkapja a szétválasztott adatot és továbbítja a subscriber-eknek.
- Ha a broker-subscriber kapcsolat megszakad, az üzenet a broker-nél elmentésre kerül és a kapcsolat helyreállásakor újra továbbításra kerül.
- A publisher-eknél, ha a broker kapcsolata értesítés nélkül megszakad, a broker saját maga tárolja az üzenetet a kapcsolódó subscriber-eknek.
- Mivel az MQTT eseményvezérelt, nem támogatja a folyamatos adatátvitelt és kontroll alatt tartja azt.
    - Az adatok csak akkor kerülnek továbbításra, amikor szükség van rájuk.



MQTT használati esetek

- Remote monitoring alkalmazások fejlesztése.
- Olyan célokra használják, mint a veszélyekre való figyelmeztetés, tűzérzékelők, lopásérzékelés vagy egy cél követése.
- Valós idejű kommunikációs alkalmazások fejlesztésére is tökéletes, ilyen például a Facebook Messenger is.
    - Lightweight/”könnyű”, nem fogyasztja annyira a telefon akkumulátorját és gyorsan kézbesíti az üzeneteket.

MQTT session/munkamenet szakaszai

- **Connection:**
  - Ha egy MQTT kliens-broker TCP/IP kapcsolatot létesít, akkor indul el ez a session.
  - A feladatot egy szabványos vagy egyéni port használatával hajtják végre.
  - Legfontosabb, hogy biztosítani kell, hogy a TCP/IP-n ne fusson régi session.
    - Ha ez megtörténik, akkor a kapcsolat megszakad.
- **Authentication:**
  - A kapcsolat befejezése előtt a kliens ellenőrzi a szerver tanúsítványának hitelességét és jóváhagyja azt.
    - Ehhez a kliens megadja az SSL/TLS tanúsítvány adatait a broker-nek, aki ellenőrzi a kiszolgálói tanúsítvány adatait.
  - Ha az SSL/TLS nem kínálja fel a szerver tanúsítványt, akkor a felhasználók ellenőrzése vagy a hitelesítés a felhasználói hitelesítő adatokon keresztül történik, amiket plain-text formában küldenek el.
    - Ha az open-source broker-ek anonim klienseket fogadnak el, akkor a felhasználónév és jelszó szakaszban nem kínálnak beviteli adatokat.
- **Communication:**
  - Miután a hitelesítés befejeződött, az MQTT session eléri a kommunikációs szakaszt, amiben a kliensek engedélyezik a subscribe-ot és a pingelést, valamint az üzenetek/műveletek közzétetelét.
  - Az MQTT legfeljebb 256 MB méretű üzenetadatokat képes továbbítani.
- **Termination:**
  - Akkor történik meg a befejezés, ha bárki, subscriber vagy publisher véget akar vetni folyamatban lévő MQTT session-nek.
  - DISCONNECT üzenetet küld, amit a broker feldolgoz.