

SZEKVENCIÁLIS ÉS ITERATÍV MODELLEK	4
HAGYOMÁNYOS TERMÉKEK VS SZOFTVERTERMÉKEK	4
<i>Szoftverek komplexitása</i>	4
<i>Szakaszok költségarányai</i>	4
A FEJLESZTÉS KÖLTSÉGARÁNYAI	4
<i>Jelentésük</i>	5
<i>Gyors szoftverfejlesztés</i>	5
<i>Szoftverkrízis</i>	5
<i>Hagyományos termék előállítás</i>	6
<i>Szoftver termékek hibaarány görbéje</i>	6
ÉLETCIKLUS MODELLEK	7
<i>Vízesés modell</i>	7
<i>V-modell</i>	8
<i>Evolúciós modell</i>	8
<i>Újrafelhasználás-orientált fejlesztés</i>	9
<i>Inkrementális fejlesztés</i>	9
<i>Inkrementális fejlesztés pipeline-alapú</i>	10
<i>Inkrementális fejlesztés előnyei</i>	10
<i>RAD modell</i>	10
<i>Cowboy Coding</i>	11
AGILIS MODELLEK	12
SCRUM	12
<i>Scrum csapatok, mint apró startupok</i>	12
<i>Scrum szerepkörök</i>	13
<i>Entitások</i>	13
<i>Indító meetingek</i>	14
<i>Folyamatos meetingek</i>	14
<i>Lezáró meetingek</i>	14
HAGYOMÁNYOS VS AGILIS MÓDSZEREK	15
GIT MULTIBRANCH KÖRNYEZETBEN	16
BRANCHEK HASZNÁLATA	16
<i>Branchek</i>	16
<i>Repository belső felépítése</i>	16
<i>Branch létrehozás</i>	16
<i>Branch váltás</i>	16
NAPI FEJLESZTÉSI RUTIN	17
<i>Merge</i>	17
<i>Branchek listázása</i>	17
<i>Branching workflows</i>	17
UML	17
UML DIAGRAMOK CÉLJA	17
<i>Mi a baj az eddigi modellekkel?</i>	17
<i>UM diagram</i>	18
<i>UML használata</i>	18
<i>UML diagram típusok</i>	18
VISELKEDÉSI DIAGRAMOK	19
<i>Use-case diagram</i>	19
<i>Activity diagram</i>	19
<i>State diagram</i>	20

<i>Interakció diagramok</i>	21
Communication diagram	21
Timing diagram	21
Sequence diagram	22
STRUKTURÁLIS DIAGRAMOK	23
<i>Deployment diagram</i>	23
<i>Belső struktúra ábrázolása</i>	23
<i>Package diagram</i>	24
<i>Component diagram</i>	24
<i>Class diagram</i>	25
OO kapcsolatok.....	25
NEM UML DIAGRAMOK.....	26
<i>Gantt diagram</i>	26
<i>Wireframe</i>	26
<i>Entity-Relation diagram</i>	27
<i>Table structure diagram</i>	27
<i>Szoftvertervezés egy lehetséges menete</i>	28
GOF CREATIONAL PATTERNS	28
TERVEZÉSI MINTÁK.....	28
<i>SOLID elvek</i>	28
<i>Architektúrális tervezési minták</i>	29
MVC workflow	29
Hamis MVVM	30
MVVM.....	30
MVVMC v2.....	31
<i>GOF minták</i>	31
Refaktorálás	31
<i>Factory method</i>	32
<i>Abstract Factory</i>	32
<i>Builder</i>	32
<i>Singleton</i>	32
<i>Prototype</i>	33
GOF STRUCTURAL PATTERNS	33
ADAPTER	33
BRIDGE	34
COMPOSITE	34
FLYWEIGHT.....	34
FACADE	35
PROXY	35
DECORATOR	35
GOF BEHAVIORAL PATTERNS	36
ITERATOR	36
CHAIN OF RESPONSIBILITY I.	36
CHAIN OF RESPONSIBILITY II.	36
VISITOR	37
OBSERVER	37
COMMAND.....	37
MEDIATOR	37
STRATEGY	38
TEMPLATE METHOD	38
MEMENTO	38

STATE.....	39
INTERPRETER	39
DOMAIN ALAPÚ TERVEZÉS ÉS MIKROSZOLGÁLTATÁSOK.....	39
ASPECT-EK	39
ONION ARCHITECTURE	40
HEXAGON ARCHITECTURE	40
DOMAIN-DRIVEN DESIGN	41
SERVICE LAYER	41
ÍRÁS VS OLVASÁS.....	42
CQRS	42
OLVASÁS GYORSÍTÁSA.....	42
CQRS HIBALEHETŐSÉGEK	43
EVENT SOURCING	43
EVENT SOURCING + CQRS + DDD	43
MONOLITIKUS ALKALMAZÁS.....	44
MIKROSZOLGÁLTATÁSOK	44
MIKROSZOLGÁLTATÁS FELÉPÍTÉSE	45
MIKROSZOLGÁLTATÁS KOMMUNIKÁCIÓ.....	45

Szekvenciális és iteratív modellek

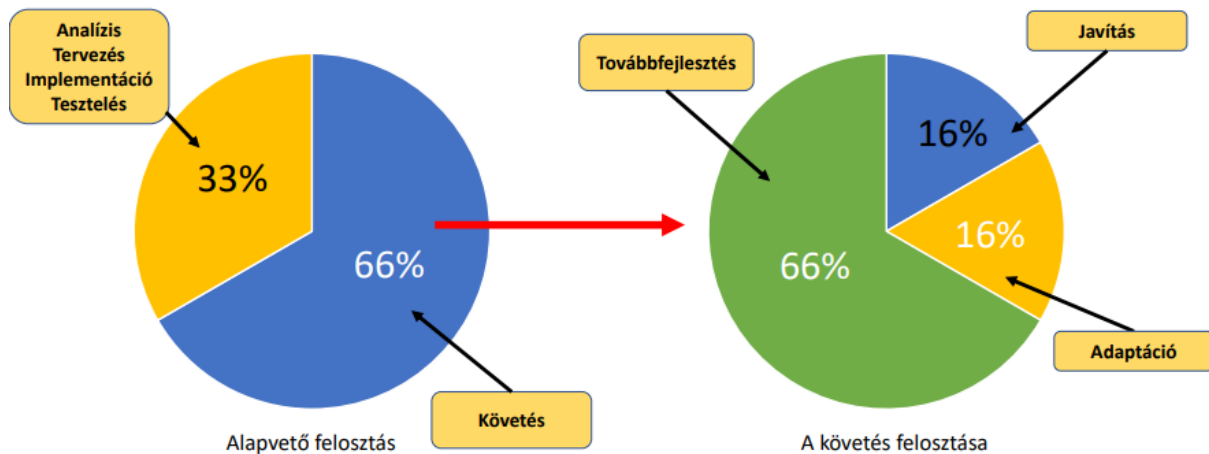
Hagyományos termékek vs szoftvertermékek

Szoftverek komplexitása

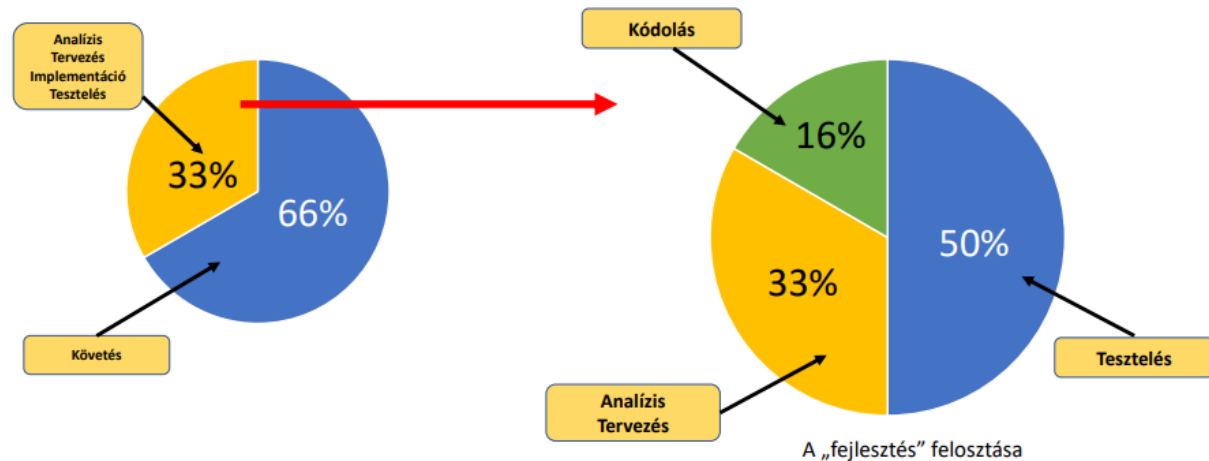
- A szoftver méretével minimum négyzetesen növekszik annak komplexitása.

Szakaszok költségarányai

- A világon megrendelt szoftverfejlesztési projektek költségarányai.



A fejlesztés költségarányai



Jelentésük

- Analízis
 - o A probléma megértése a megrendelő elmondása alapján
 - o Specifikáció megfogalmazása
- Tervezés
 - o Specifikációból adatbázis terv, rendszerterv, OOP osztályok tervezése, rétegek kialakítása, stb.
- Implementáció
 - o A tervezés alapján a szoftver lekódolása
- Tesztelés
 - o Az elkészült szoftver letesztelése
 - Nem utólag, hanem TDD módszerrel.
- Javítás
 - o Fejlesztői részről minden tökéletesen működik, de kibuknak a hibák
 - Rossz specifikáció, stb.
- Adaptáció
 - o Megrendelő adottságaihoz alakítás
 - Ügyélnek már van szervere, levelezése, adatbázisa, stb.

Gyors szoftverfejlesztés

- Kizárólag csapatmunkában
- Multibranch GIT környezetben
- Projektmenedzsment tool-ok támogatásával
- Újrahasznosított komponensekkel
- Frameworkök használatával
- Folyamatos teszteléssel
- Folyamatos integrációval
- Jól bevált, profik által javasolt kódmintákkal
- Folyamatos vevői véleményeztetéssel
- Fenntartható, moduláris kód írásával

Szoftverkrízis

- Kor jellemzői
 - o Gyenge hardver
 - o Fejlesztői eszközök nem voltak
 - Text editor + compiler
 - o Monolitikus programozás (spagetti kód)
 - o Team munkához semmilyen eszköz nincsen (manual merge)
 - o Konfiguráció változást nem támogatja a szoftver
- Megoldás
 - o Új paradigmák (pl.: OOP) modularizálhatóság megjelenése

Hagyományos termék előállítása



- Gyerekbetegségek
 - o Tervezési, gyártási hibák
 - o Hibás alapanyagok
- Elkopás okai
 - o Öregedés, szerkezeti elváltozások



Szoftver termékek hibaarány görbéje

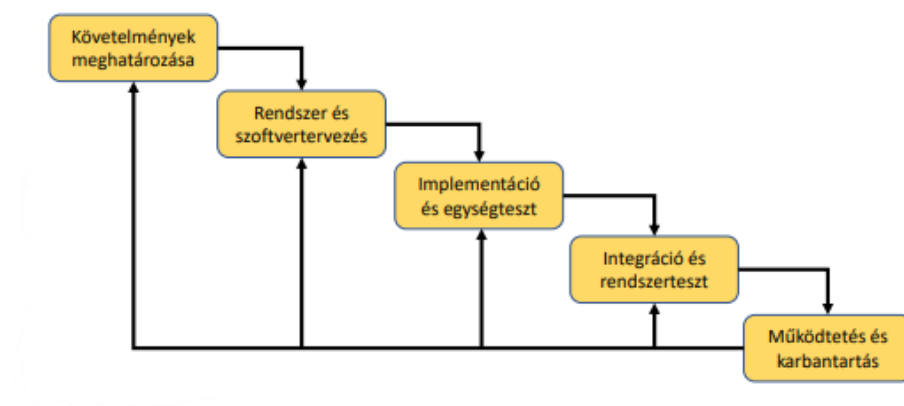
<p>A graph showing the error rate (hibaarány) on the y-axis versus time (idő) on the x-axis. The curve starts high and decays exponentially towards the x-axis. A callout box labeled 'Konstans az elavulásig' (Constant until obsolescence) points to the curve.</p>	<p>A graph showing the error rate (hibaarány) on the y-axis versus time (idő) on the x-axis. The curve starts high and decays, but it has several sharp vertical spikes. A callout box labeled 'Változtatások' (Changes) points to one of these spikes.</p>
<ul style="list-style-type: none">- Nincsen öregedés	<ul style="list-style-type: none">- Van helyette állandó módosítási igény- Az újabb funkciók, újabb hibákat eredményeznek- Hibák összeadódnak

Életciklus modellek

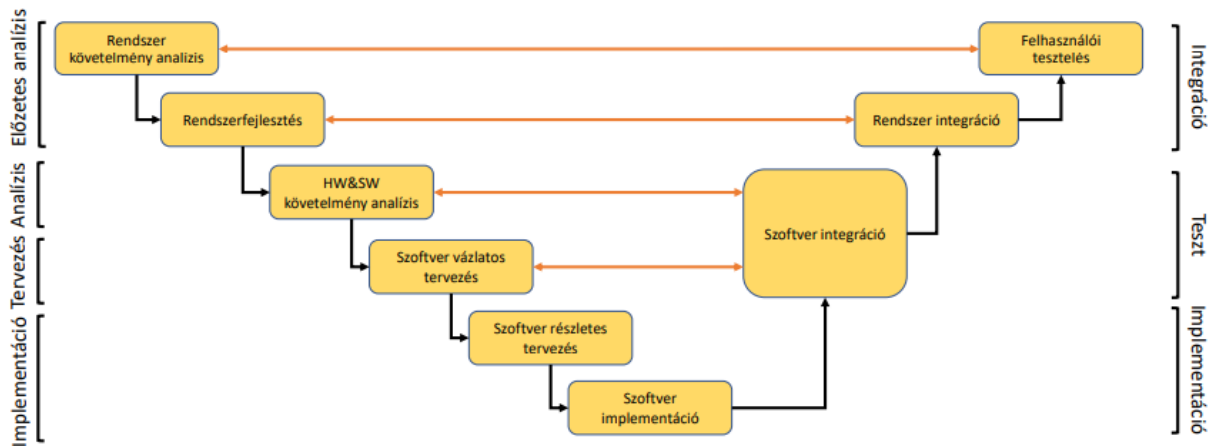
- Szekvenciális modellek
 - o Vízésés modell
 - o V-modell
- Iteratív/inkrementális modellek
 - o Evolúciós modell
 - o Formális rendszerfejlesztés
 - o Újrafelhasználás-orientált fejlesztés
 - o Inkrementális fejlesztés
 - o RAD modell
 - o Spirál modell
- Agilis modellek
 - o XP
 - Extreme Programming
 - o SCRUM
 - o Kanban
- Rational Unified Process (RUP)

Vízésés modell

- 1970: W. W. Royce
- Hagyományos mérnöki szemlélet követése
- Elterjedt modell, mert már régi.
- Problémái
 - o Valós projektek nem így működnek.
 - o Minden a specifikáció minőségétől függ.
 - o Nagyon későn lát a megrendelő működő programot.
 - o Kezdeti bizonytalanságot nehezen kezeli.
 - o Tesztelés szerepe nem eléggé hangsúlyos.

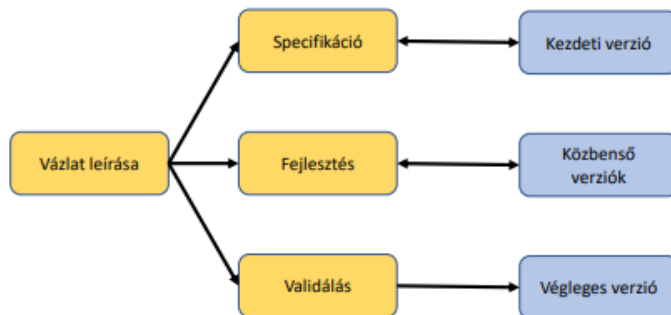


V-modell



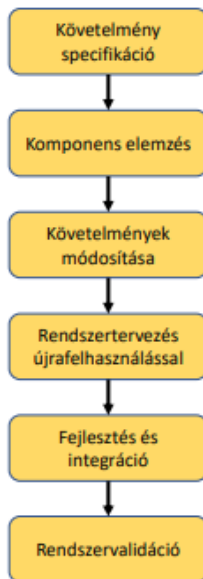
- 1991: német védelmi minisztérium
- Vízses modell továbbfejlesztése a tesztelés hangsúlyossá tételével.
- Szigorú dokumentálás
- Nem küszöböli ki a vízses modell problémáit.
- Kritikus rendszerek fejlesztésénél előnyös.

Evolúciós modell



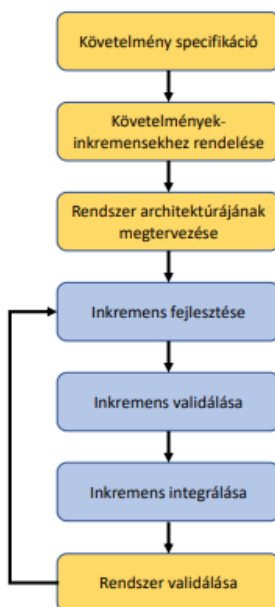
- Kifejlesztünk egy kezdeti implementációt.
- Véleményeztetjük a felhasználókkal.
- Sok verzió keresztül finomítjuk, amíg el nem érjük a kívánt rendszert.
- Problémák:
 - o Nehezen menedzselhető a folyamat.
 - o Minőségbiztosítási problémák.
 - o Speciális eszközök és technikák igénye.

Újrafelhasználás-orientált fejlesztés



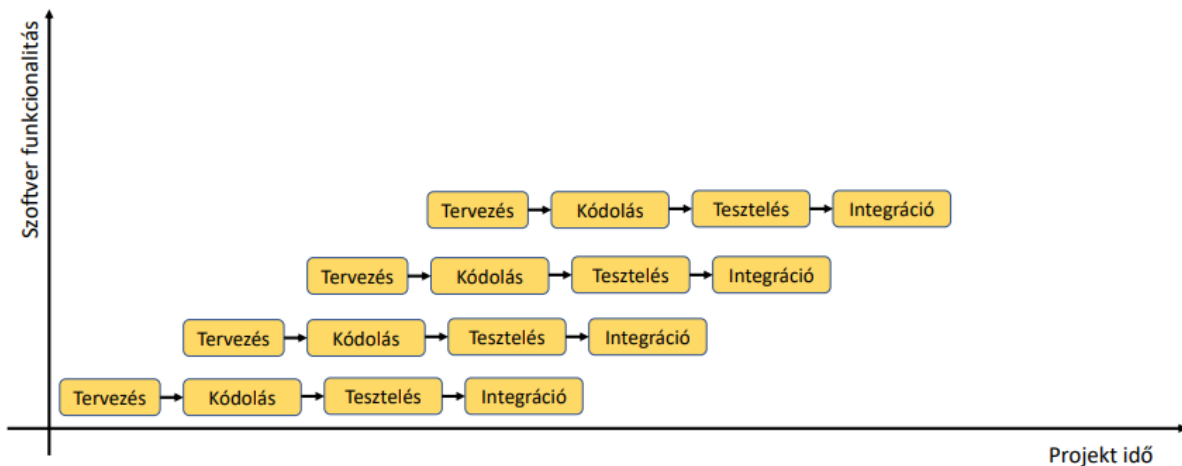
- A fejlesztési idő nagy mértékben lerövidíthető.
- Lényegesen olcsóbbá válik a fejlesztés.
- A rendszer minősége és megbízhatósága jobb, mert ellenőrzött komponenseket építünk be.
- Nagy library gyűjtemény esetén nehéz a pontos komponens kiválasztása.

Inkrementális fejlesztés



- A rendszert kisebb egységekre bontjuk
- Minden egység önálló fejlesztés – validálás – integrálás folyamatot kap.
- Minden inkremens külön egyeztetett, elkészülte után a megrendelő használatba veheti.
- Cél a komplexitás csökkentése.
- Megrendelőnek lehetősége van bizonyos követelményekkel kapcsolatos döntések elnapolására.
- Inkremens lehet például az alrendszerek.

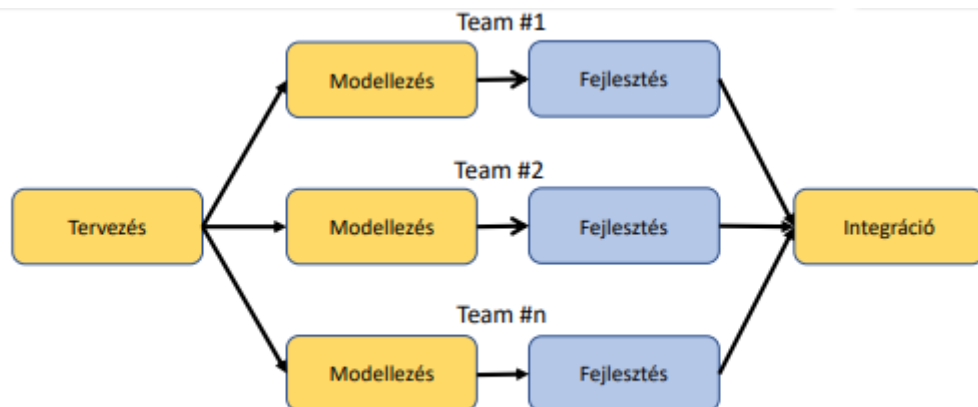
Inkrementális fejlesztés | pipeline-alapú



Inkrementális fejlesztés előnyei

- A szoftver már menetközben használhatóvá válik a megrendelő számára.
- Kritikus követelmények teljesülnek először.
- Kisebb a kockázata a projekt kudarcának, biztonságos fejlesztési koncepció.
- A legkritikusabb funkciókat teszteljük legelőször.

RAD modell



- Rapid Application Development
- Vízesés modell high-speed adaptációja
- Építőelemei
 - o Újrafelhasználás-orientált fejlesztés
 - o Komponensekre bontás
 - o Kódgenerálás
- Sokoldalú, párhuzamosan dolgozó csapatok
- Megrendelő szakemberei is a csapatok részei lehetnek.
- Hátrányai
 - o Nagy projekt
 - Hatalmas humán erőforrás igény
 - o Ha nehezen modularizálható a rendszer, akkor nem működik.

Cowboy Coding

- A kódírás előtérbe helyezése az összes többi fázissal szemben.
- Előnyök
 - Nagy szabadságfok a megoldásnál
 - Válságkezelést gyorsan orvosolja
 - Nem köti az architekturális terv a programozót.
- Hátrányok
 - Team-munka lehetetlen
 - Nincs minőségbiztosítás
 - Nem fenntartható a kód
 - Gyenge dokumentáció
 - Nagy projekt esetén nem használható
 - Nem lehet vele jól keresni

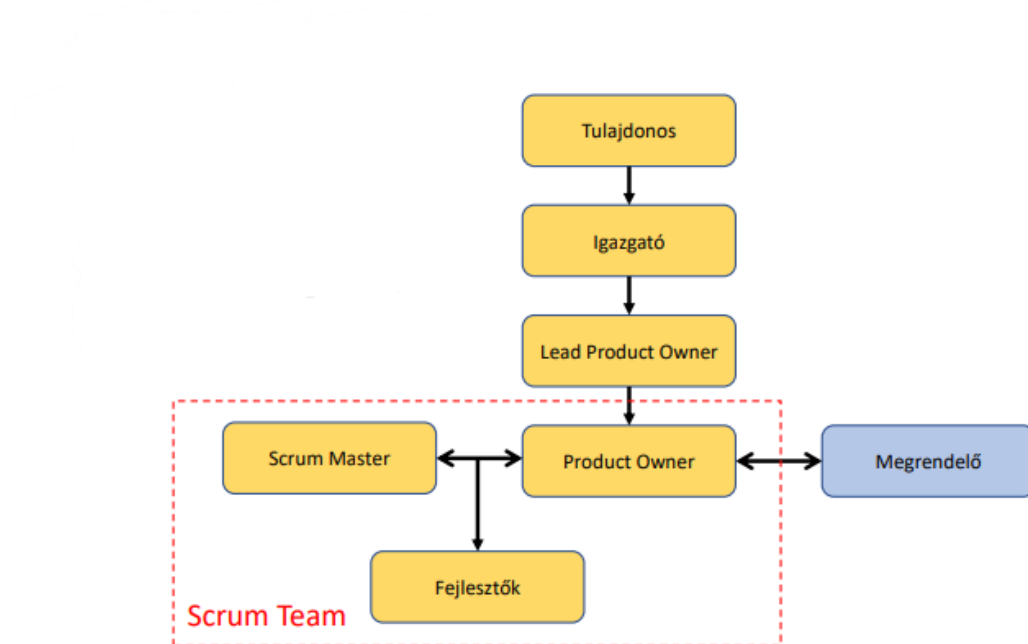
Agilis modellek

Scrum

- 1986
 - o Hirotaka Takeuchi és Ikujiro Nonaka
- Sikeres vállalatok tanulmányozásából született a Scrum
- Szerepkörök
 - o Disznók – Scrum Master / Product Owner / Team
 - Szinte önfeláldozás az elvárt (disznó önmagát adja az ételbe)
 - o Csirkék – Menedzserek / tulajdonosok / haszonélvezők

Scrum teamek, mint apró startupok

- Vezetőség az erőforrásokat adja
 - o Emberi
 - o Anyagi
 - o Infrastrukturális
- Adott Scrum Team közvetlenül kommunikál a megrendelővel
- Teljesen az ő felelősségük a projekt esetleges kudarca



Scrum szerepkörök

- **Product Owner**
 - A csapat tagja, a saját cégünk embere, kollégánk
 - Megrendelő érdekeit képviseli
 - Tisztában van a megrendelő céljaival/problémaival
 - Tisztában van a prioritásokkal
 - Megszervezi a rendszeres demókat
 - Nem fejlesztő
 - Érti a gazdasági folyamatokat, érti az IT-t
 - Product Backlog kezelője
- **Scrum Master**
 - Csapat tagja, saját cégünk embere
 - Hagyományos projektmenedzser szereppel egyezik a feladata
 - Felügyeli a folyamatokat, betartattja a Scrum szabályait
 - Konfliktusokat kezel, akadályok elhárítását irányítja
 - A meetingeket ő szervezi, ő vezeti
 - A Scrum csapat feje
- **Scrum Team**
 - 5-9 fő alkotja
 - Szükséges:
 - Elemző
 - Fejlesztő
 - Tesztelő
 - Matematikus
 - Ők végzik a tényleges fejlesztést
 - Felelőségük, hogy egy sprintre bevállalt feladatokat elvégezzék
 - Fejlesztői fokozatok
 - Mindenki egyenlő
 - Junior > Medior > Senior
 - Tesztelő
 - Mindenki a saját kódját teszteli
 - Unit tesztek után peer review más fejlesztővel
 - Van manuális teszt, amit nem a fejlesztők végeznek

Entitások

- **User Story, Task**
 - Egy 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 Story-k
 - A sprint vége egy leszállítható szoftver
- **Product Backlog**
 - Még elkészítésre váró Story-k gyűjtőhelye
 - PO tartja karban (prioritásokat rendel a story-khoz)
 - **PO:** üzleti érték becslése
 - **Team:** ráfordítás becslése
 - **ROI:** Return of Investment = üzleti érték / ráfordítás

- **Sprint Backlog**
 - o A Product Backlog de csak az adott sprintre bevállalt storykra szűrve.
- **Burn down/up chart**
 - o Napi eredmény diagram
 - o Megmutatja, hogy a csapat mennyire tartja az eredeti ütemtervet
- **Impediment**
 - o Akadály, ami a munkát hátráltatja
 - o Tipikusan valamilyen munkahelyi probléma
 - o Scrum Master feladata elhárítani

Indító meetingek

- **Sprint (pre-)grooming**
 - o Architect(ek) + Scrum Master
 - o Storyk ellenőrzése, pontozása (nehézség/idő > Fibonacci számokkal)
- **Sprint planning**
 - o Team bevállal storykat (product backlog > sprint backlog)
 - o Figyelembe veszik a pontozást

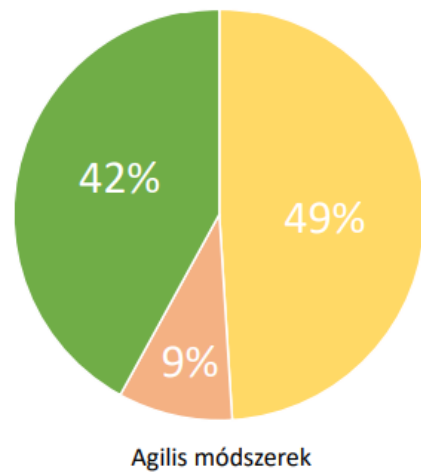
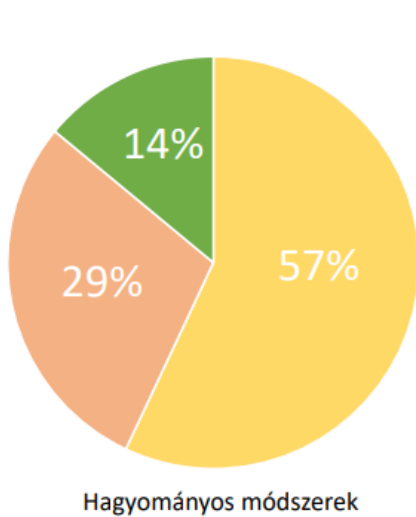
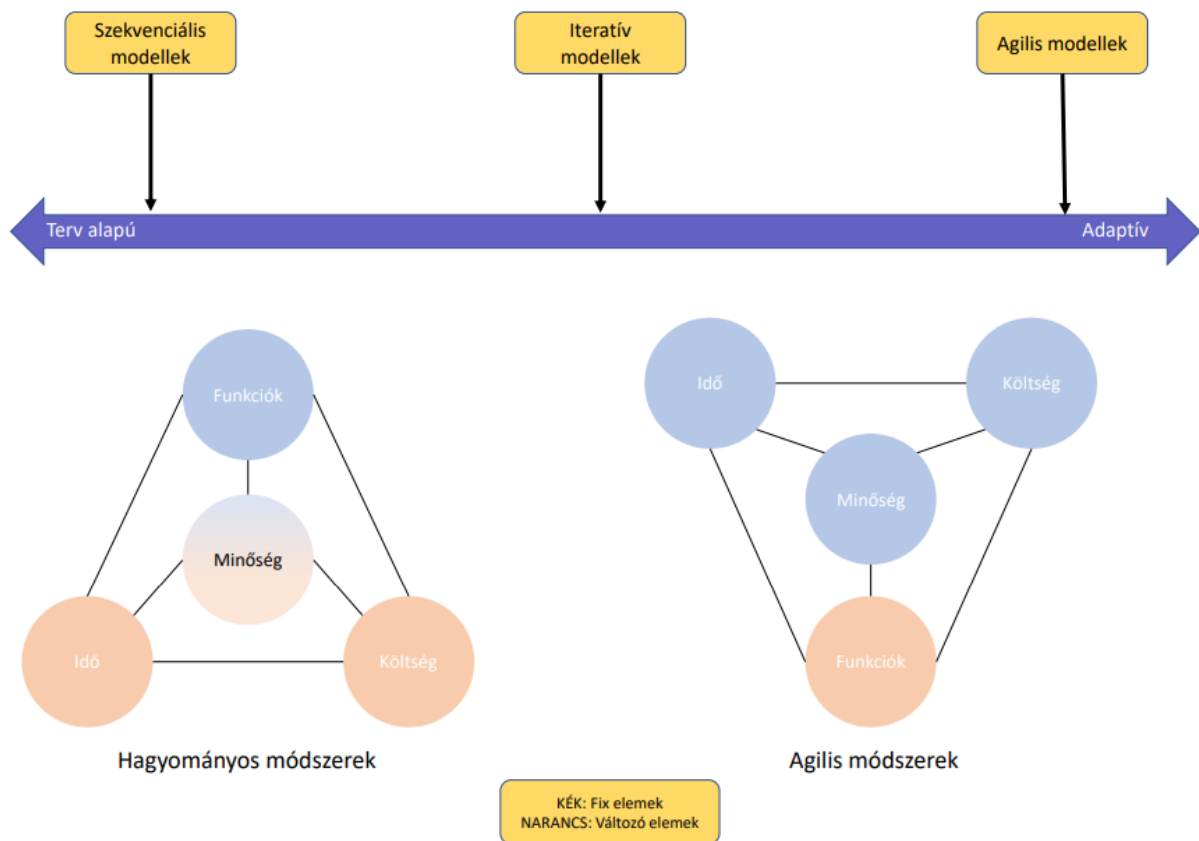
Folyamaos meetingek

- **Daily stand-up**
 - o Minden nap ugyanakkor
 - o Egész team részt vesz
 - o Kb 15 perc, állva, eszközök nélkül, szóban
 - o Mit csináltam tegnap? Mit fogok csinálni ma? Milyen problémám van?
- **Sprint refinement**
 - o Hetente 1 maximum (3 hetes sprintben maximum 2 alkalom)
 - o Storyk/Taskok áttekintése
 - o Gyors hibaelhárítások
 - o Nem része a Scrumnak, de alkalmazzák
- **Egyéb meeting**
 - o Bárki bárkivel együtt dolgozik egy problémán, csapat részhalmaza egyeztet

Lezáró meetingek

- **Sprint review**
 - o Team + Scrum Master + Product Owner
 - o „User acceptance test”
 - o Eredmények bemutatása, PO dönti el, hogy sikeres-e a sprint
 - o Kimaradt Storyk/Taskok, „next”-be mennek
 - Következő sprintre
- **Sprint retrospect**
 - o Review után tartják, Team + Scrum Master
 - o Személyes tapasztalatok/javaslatok megvitatása
 - o Emeljük ki 3 dolgot
 - Pöttyözzük az összes ember 3 dolga közül a 3 legfontosabbat
 - o Céges szintű problémákat SM továbbítja felfelé
 - o Személyes konfliktusok megbeszélése

Hagyományos vs Agilis módszerek



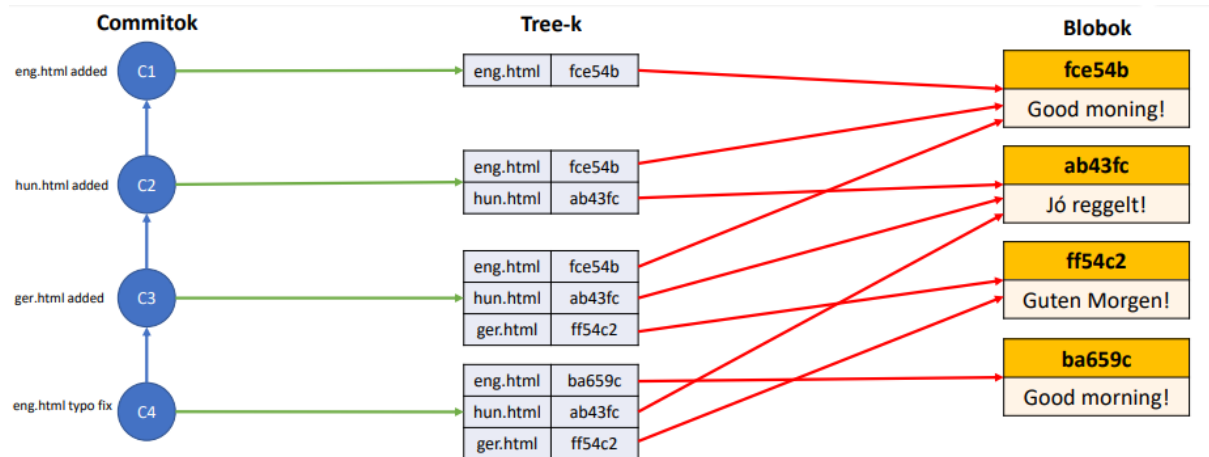
GIT multibranch környezetben

Branchek használata

Branchek

- Cél
 - o Eltérünk a fejlesztés menetétől, anélkül, hogy elrontanánk.
- Branch
 - o Egy pointer, ami egy commitra mutat.
 - o Alapból létezik egy master
 - o Minden commitolásnál egyet arrébb megy
- HEAD
 - o Egy pointer, ami az aktuális local commitra mutat.
 - o A working directory ebben az állapotban van.

Repository belső felépítése



Commit tartalma: referencia az ősré/ősökre és a tree-re

Branch létrehozás

- Legyen egy testing nevű branch is a master mellett
 - o **git branch testing**
- Az aktuális állapotba helyezi a testing mutatót.

Branch váltás

- Átállás másik branchre
 - o **git checkout testing**
- Branch létrehozás és checkout egyben
 - o **git checkout -b testing**
- A workdir nem változik meg, mert jelenleg mindkét branch ugyanoda mutat.
- El lehet indulni két irányba a két branch-el.
- A testingen más kód
- Masteren haladunk tovább
- Log parancs
 - o **git log --oneline --decorate --graph --all**

Napi fejlesztési rutin

- Szükség van egy új feature-re
 - o Leágazunk a fő branchről egy új feature branchhel
- Kapunk egy hotfix kérést
 - o Leágazunk a fő branchről egy új hotfix branchhel
 - o Itt megoldjuk
 - o Tesztelés után merge a fő branchre
- Fő branch
 - o master
 - o release verziók ide kerülnek fel

Merge

- Conflictok feloldását nem csak kézzel tehetjük meg
- A merge és add parancsok között futtatjuk

Branchek listázása

- Összes branch listázása
 - o **git branch**
- Összes branch listázása + utolsó commit message-ek
 - o **git branch -v**
- Szűrő feltételek
 - o **git branch --merged**
 - o **git branch --no-merged**

Branching workflows

- **Long-running branches/GitFlow**
 - o **Long-running branches**
 - A master mindig stabill
 - Masteren lévő merge commitok release pontok

UML

UML diagramok célja

Mi a baj az eddigi modellekkel?

- Kommunikációs szakadék van a megrendelő és a fejlesztők között
 - o Agilis módszerek bevonása
 - o Prototípusok fejlesztése, véleményezése
 - o Nincs még közös nyelv
- Kommunikációs szakadék van fejlesztő és fejlesztő között
 - o Tapasztalattól, tudásszinttől függően is előfordul, hogy nem egy nyelvet beszélnek.
 - o UM Diagrammok használata

UM diagram

- Unified Modelling Language
 - o Grafikus leírónyelv, ami segíti az alábbi folyamatokat
 - Vizualizálás
 - Specifikálás
 - Tervezés
 - Dokumentálás
- Kinek jó ez?
 - o Megrendelő egy folyamatábrát könnyen tud értelmezni.
 - o Fejlesztő könnyebben megérti, hogy a másik fejlesztő rendszere hogy működik.
 - o Vizuális ábrázolás jobb megértést biztosít.
 - o Dokumentáció és kellően alapos lesz általa

UML használata

- UML egy szigorú modellező nyelv
 - o Agilis UML, ha már elhagyható minden.
- Modellező eszközök
 - o Microsoft Visio
 - o Visual Paradigm for UML
 - o Rational Rose
 - o Enterprise Architect (legelterjedtebb, de fizetős)
 - o DIA

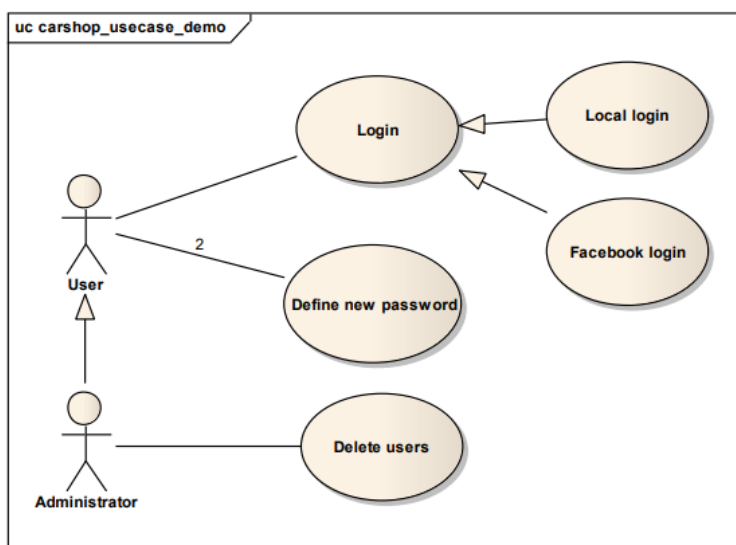
UML diagram típusok

- **Viselkedési diagramok (behavioral diagram)**
 - o Használati eset diagram (use-case diagram)
 - o Aktivitás diagram (activity diagram)
 - o Állapotgép diagram (state machine diagram)
 - o Interakciós diagram (interaction diagram)
- **Strukturális diagramok (structure diagram)**
 - o Osztály diagram (class diagram)
 - o Komponens diagram (component diagram)
 - o Objektum diagram (object diagram)
 - o Kompozit-struktúra diagram (composite structure diagram)
 - o Telepítési diagram (deployment diagram)
 - o Csomag diagram (package diagram)
 - Részletes tervezés / Megvalósítás fázis
- **Kinézetű terv (wireframe)**
 - o Nem UML
 - o Előzetes tervezési fázis

Viselkedési diagramok

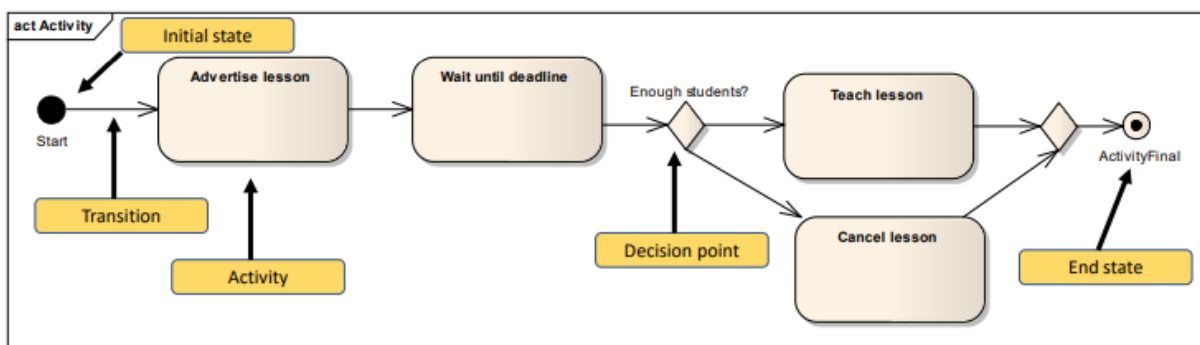
Use-case diagram

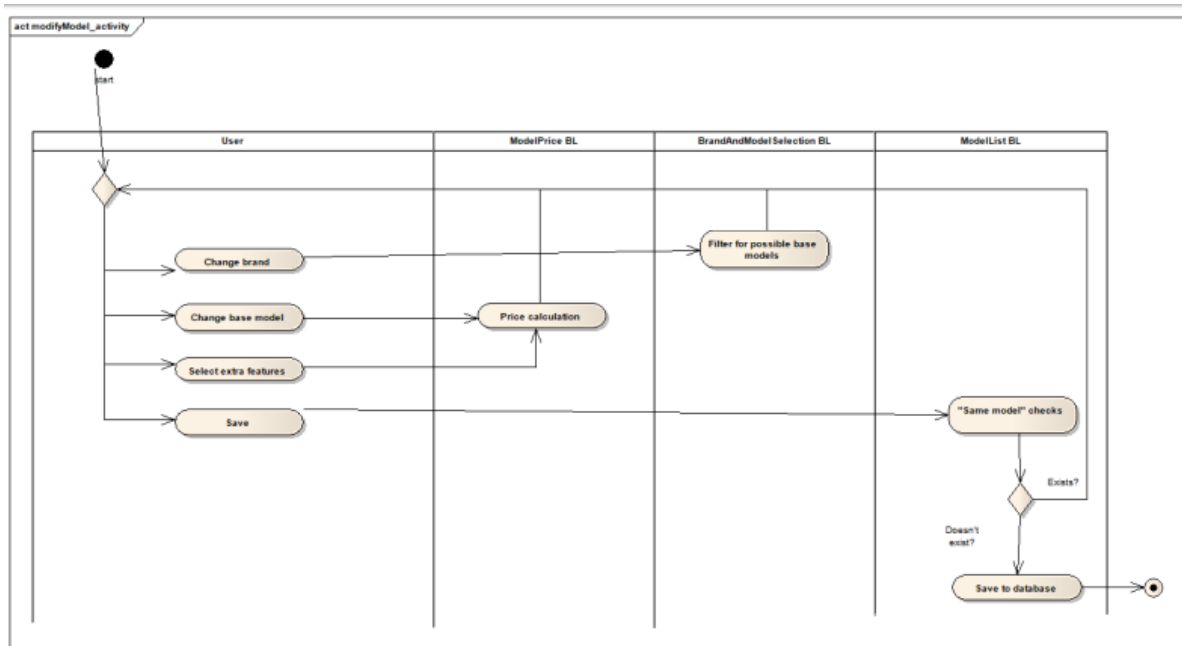
- Célja
 - o Megrendelővel való egyeztetés, hogy pontosan milyen szerepköröket, funkciókat képzelt el, és ezeket a funkciókat melyik szerepkörrel lehet igénybe venni.
- Aktor és használati eset közötti megfeleltetés
- Öröklődés engedélyezett
 - o Aktorok között
 - o Használati esetek között
- Számosság (aktor és használati eset között)
- Tartalmazás/kibővítés
 - o Használati esetek között



Activity diagram

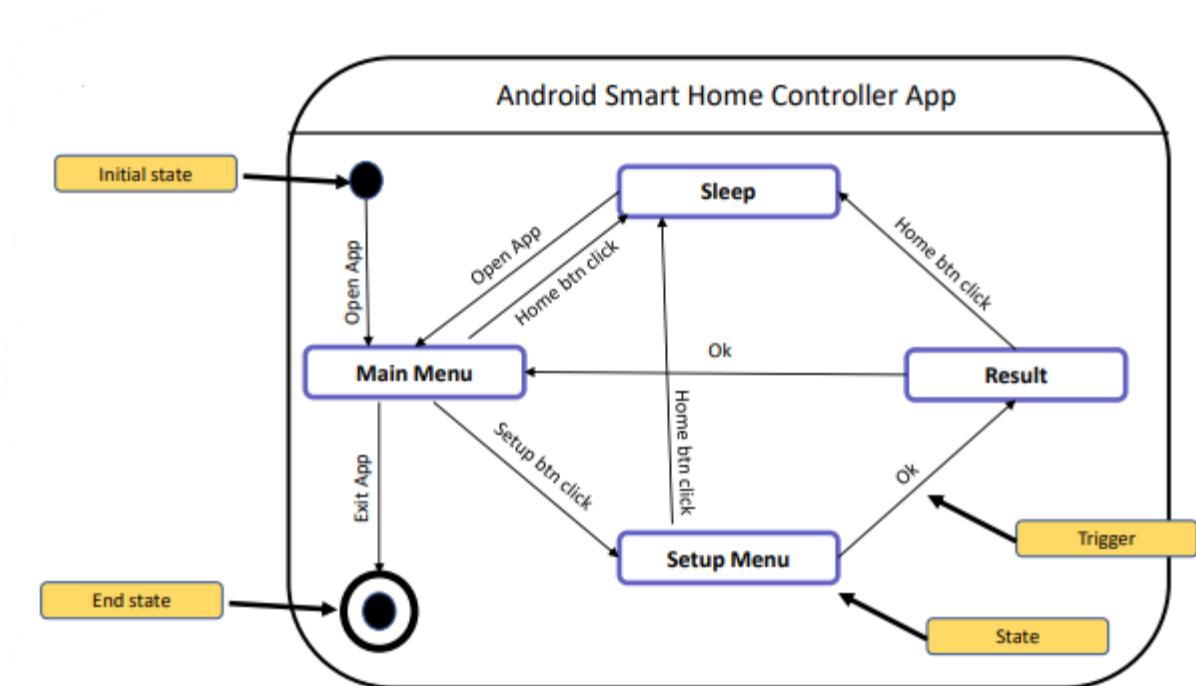
- Use-case utáni következő lépés
- Használati eseteknél a cél
 - o Belső folyamatok ábrázolása
 - o Egymás után következőségek ábrázolása
- 1 használati eset > 1 Activity diagram
- Egy diagramon belül más use-case-ek is előjöhethetnek.





State diagram

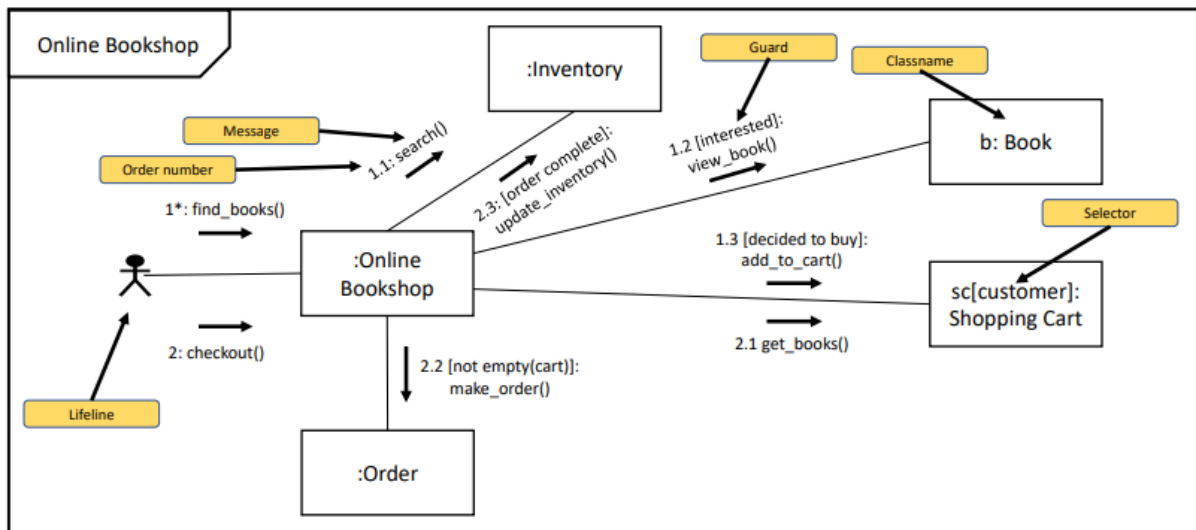
- Rendszer/objektum állapotainak egymás után következőségét ábrázolja.
- Végés állapotgépekre nem az egyetlen eszköz
 - o State transition table
 - Első oszlop: állapotok
 - Első sor: események
 - Cellák: feltételek + új állapot
 - o Petri hálók
 - o SDL state machine
- Probléma:
 - o Már ez a méret is átláthatatlan



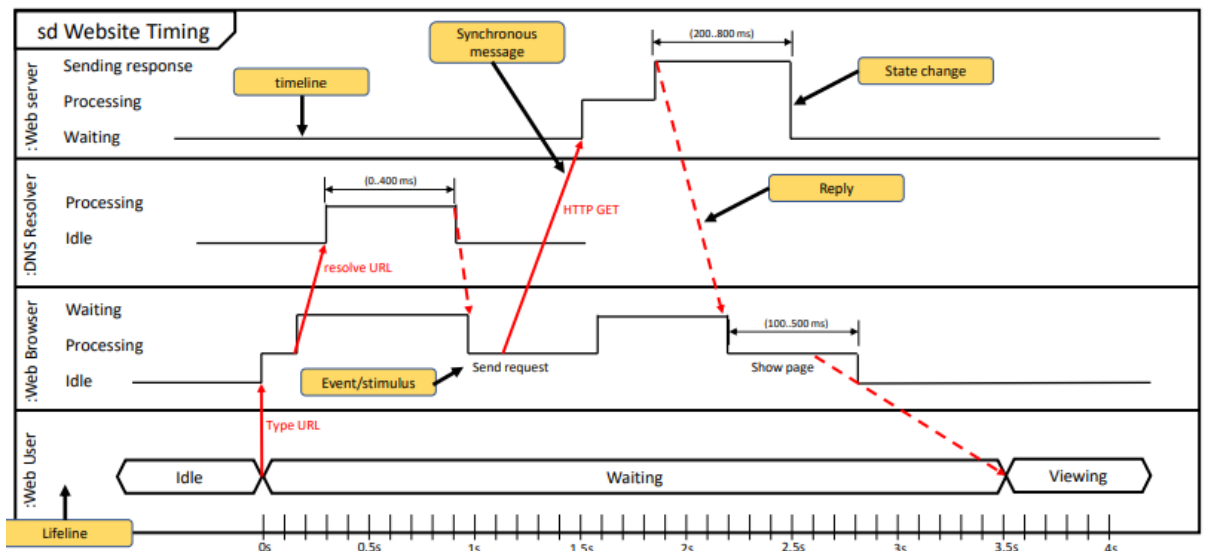
Interakciós diagramok

- Egy folyamat résztvevői (aktorok vagy modulok vagy rétegek) közötti kommunikációs folyamatokat ábrázoljuk.
- Három különböző diagramot különböztet meg
 - o Communication diagram
 - Résztvevők és sorrend
 - o Timing diagram
 - Időzíti információk, megkötések és állapotok is
 - o Sequence diagram
 - Ciklusok, feltételek és élettartamok is

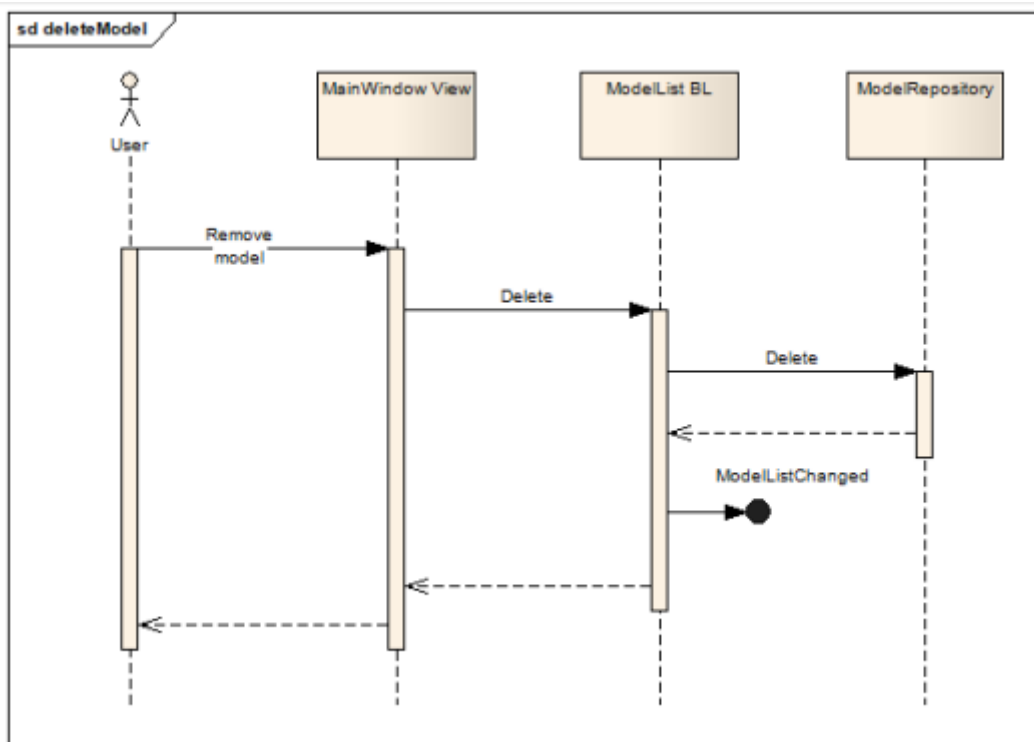
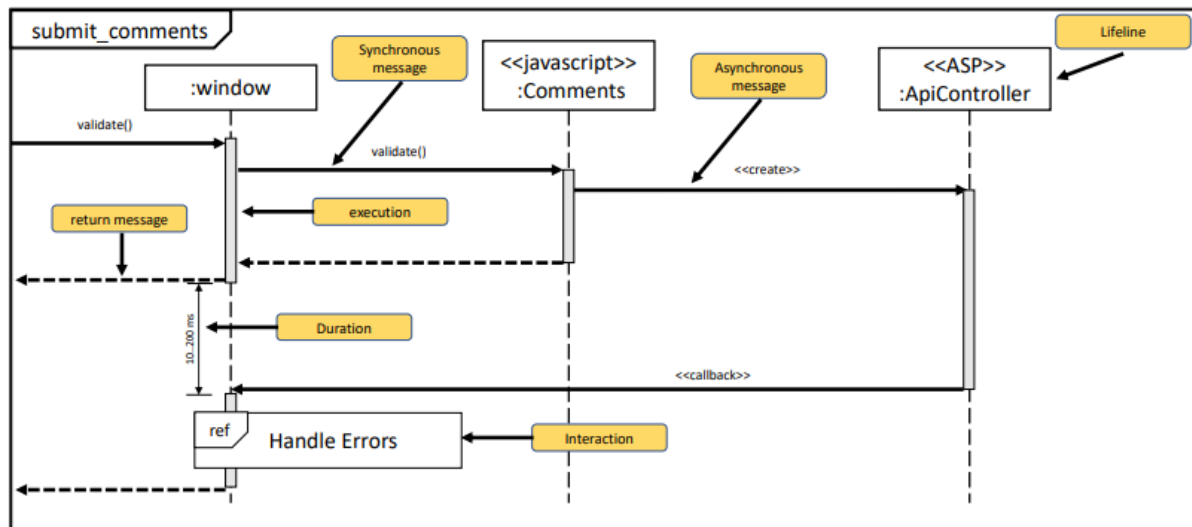
Communication diagram



Timing diagram



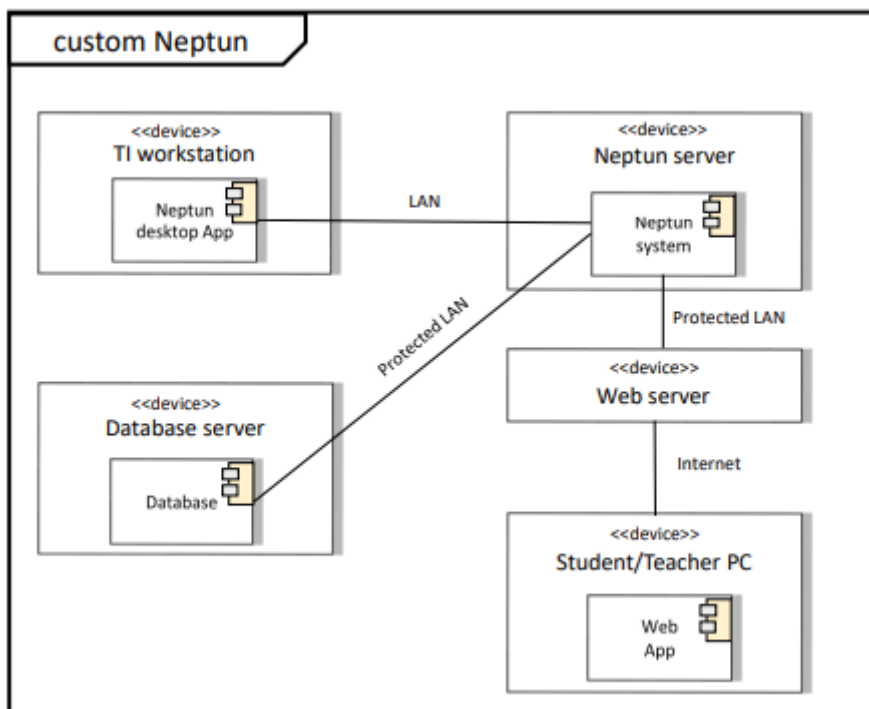
Sequence diagram



Strukturális diagramok

Deployment diagram

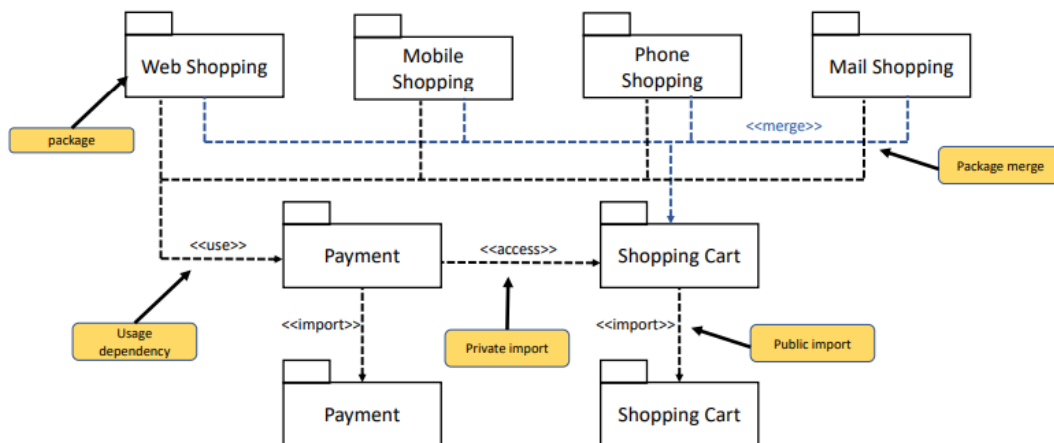
- A rendszer futásának körülményeit/elemeit bemutató diagram
- A működtető elemek lehetnek
 - o Számítógépek/kiszolgálók
 - o Hálózati csomópontok
 - o Egyéb környezetek (VM, konténer)
- Akár a fejlesztési fázis első diagramja is lehet
 - o Ha a környezet már készen van (új szoftvert kell írni meglévő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



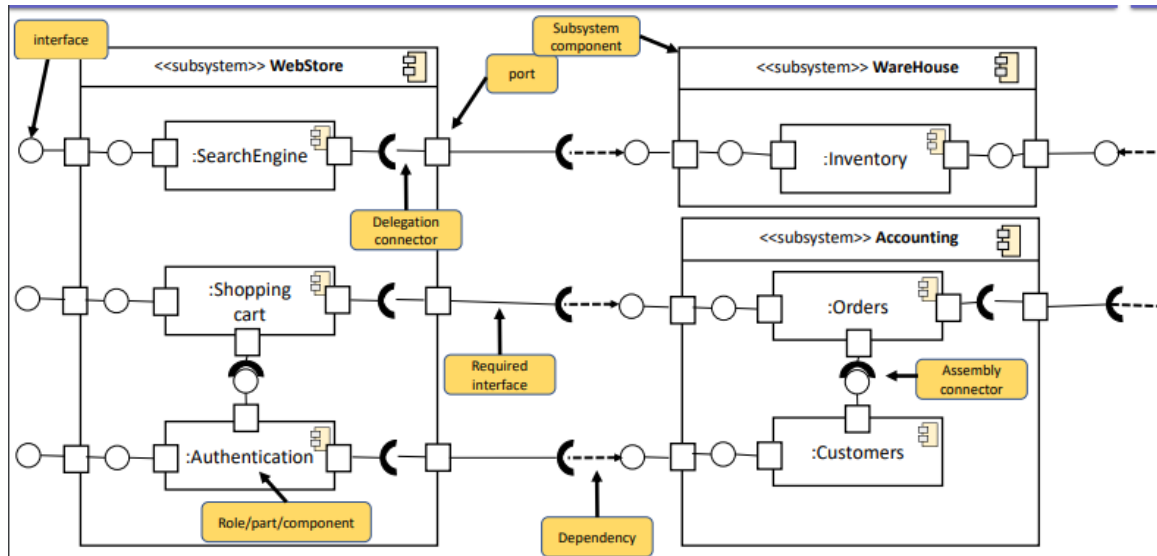
Belső struktúra ábrázolása

- Components vs Package diagram
 - o Szinte ugyanaz
- Lehetséges megközelítés
 - o Package: felépítés, tartalmazás, függőségek
 - Amiket csomagolni lehet (pl.: egy DLL-be)
 - o Component: kapcsolódások, interfészek
 - Helyi vagy távoli alrendszerek is.
- A component diagram egy szélesebb leírása, külső függőségeket is tartalmazza.

Package diagram

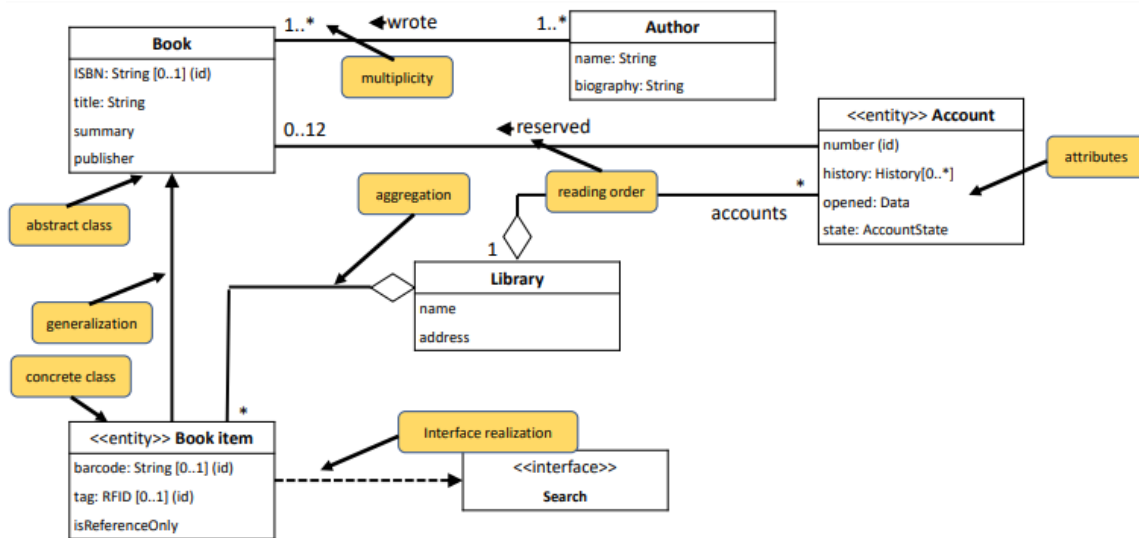


Component diagram

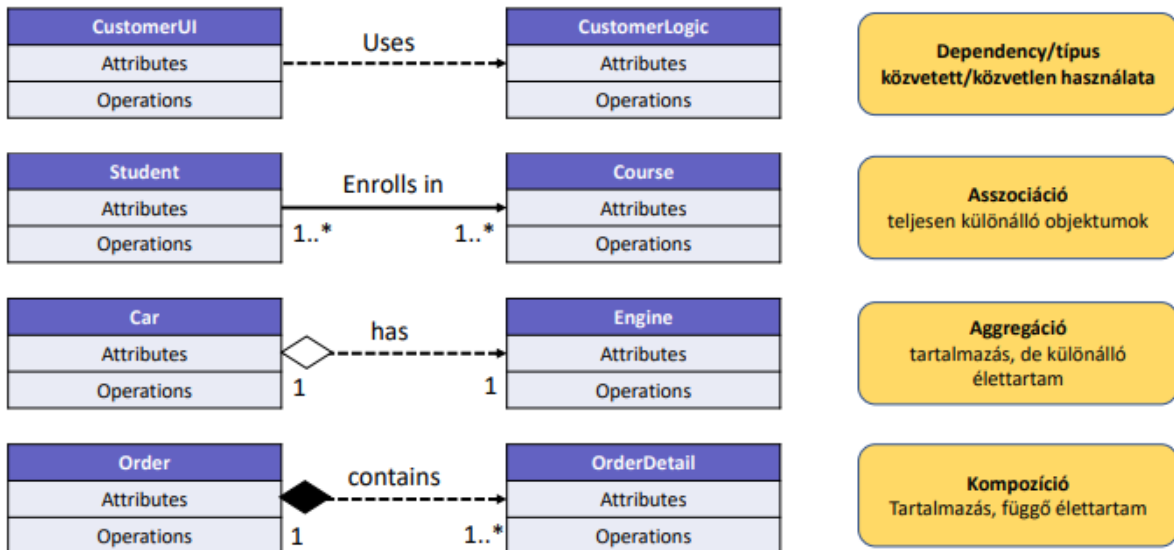


Class diagram

- Osztályok vagy konkrét objektumok ábrázolása
 - o Adattagok, metódusok
- Mikor készül el?
 - o Korai tervezéskor még nem kell
 - Package diagram interfészei elegek
 - o Pregroom meeting
 - Architect feladata is lehet akár



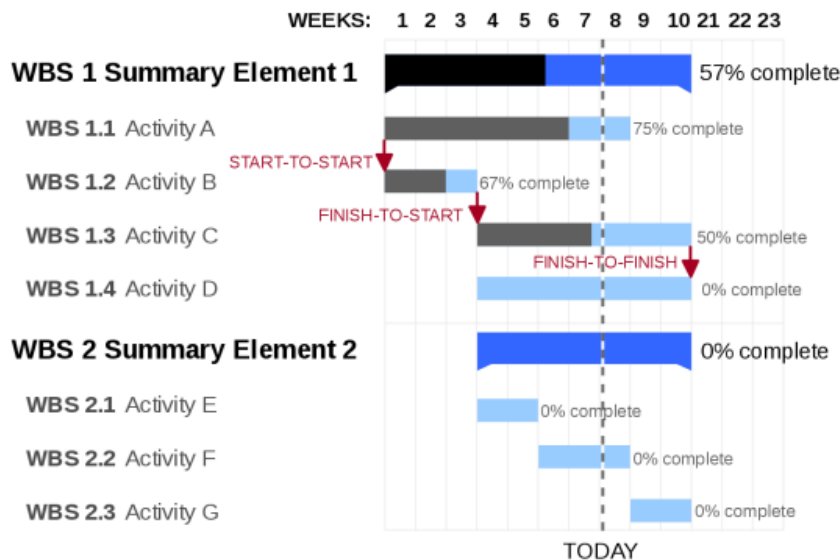
OO kapcsolatok



Nem UML diagramok

Gantt diagram

- Időterv készítése
- Egy mástól függő folyamatok ábrázolása



Wireframe

- UML létrehozásakor nem gondoltak arra, hogy a megjelenítés is a modell része lehet
- Legelső prototípus, amit oda tudunk adni a megrendelőnek
- Létrehozása
 - o Fejlesztőeszközökkel
 - WPF-ben XAML összerakása adatkötés nélkül
 - Wireframe eszközzel
 - Rajzprogrammal
 - o Folyamatok előrejelzése kötelező
 - o Melyik ablakból hogyan és miként lehet átmenni a következőbe és vissza.


Egyedi autó

Márka: Opel

Modell: Combo

Extrák:

- ☐ A
- ☒ B
- ☐ C
- ☒ D
- ☐ E
- ☐ F
- ☐ G
- ☐ H



Ár: 3.999.000

Entity-Relation diagram

- Adatmodellezéskor használjuk
- Egyedek és köztük lévő kapcsolatok tervezése
 - o Majd normalizálás
 - o Majd adatbázis készítése

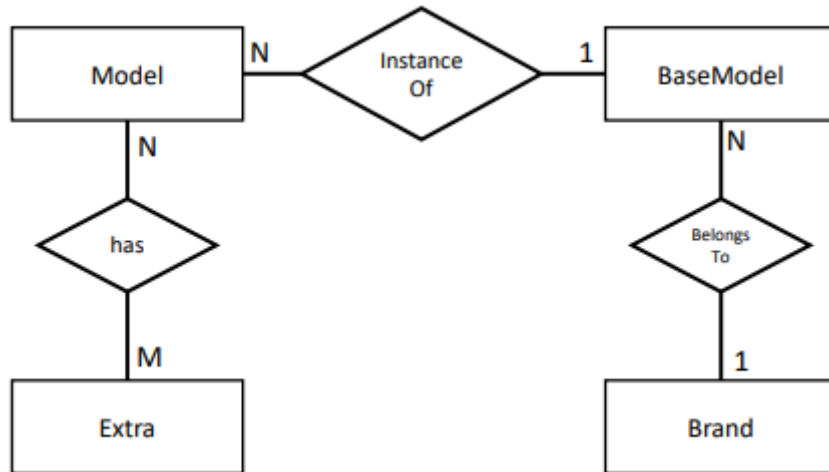
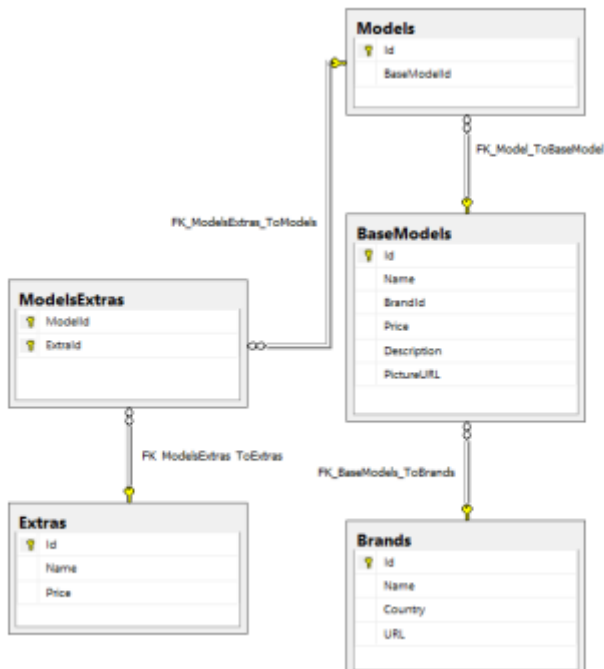


Table structure diagram

- Adatbázis táblák szemléltetése
- Elsődleges és idegen kulcsok ábrázolása
- Minden szoftverfejlesztési projekthez kell



Szoftvertervezés egy lehetséges menete

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

GOF Creational Patterns

Tervezési minták

SOLID elvek

- **Single Responsibility**
 - o Minden osztály egy dologért legyen felelős és azt jól lássa el.
 - o Ha nem követjük, akkor
 - Spagetti kód, átláthatatlanság
 - Nagy méretű objectek
 - Mindenért felelős alkalmazások és szolgáltatások
 - Alkalmazás szinten egyre kevésbé követik
 - Winamp, chrome, systemd, moodle
- **Open/Closed Principle**
 - o Egy osztály legyen nyitott a bővítésre és zárt a módosításra (nem írhatunk bele, de származtathatunk tőle). Ha nem követjük, akkor
 - Átláthatatlan, lekövethetetlen osztályhierarchiák, amik nem bővíthetőek
 - Tünetek
 - Leszármazott megírásakor módosítanunk kell az őosztályt is (TILOS)
 - Egy kis funkció hozzáadásakor több osztályt kell hozzáadni ugyanabban a hierarchiában
- **Liskov substitutable**
 - o Őosztály helyett utódpéldány legyen mindig használható
 - o Compiler supported, hiszen OOP elv (polimorfizmus)
 - o Ha egy kliensosztály eddig X osztállyal dolgozott, akkor tudnia kell X leszármazottjával is dolgoznia.
- **Interface segregation**
 - o Sok kis interfészt használjunk egy hatalmas mindent előíró interfész helyett. Ha nem követjük, akkor
 - Tünetek
 - Egy osztályt létrehozunk valamilyen céljából, megvalósítjuk az interfészt és rengeteg üres, fölösleges metódusunk lesz.
 - Az interfészhez több implementáló osztály jön létre a kód legkülönbözőbb helyein, más-más részfunkcionalitással.

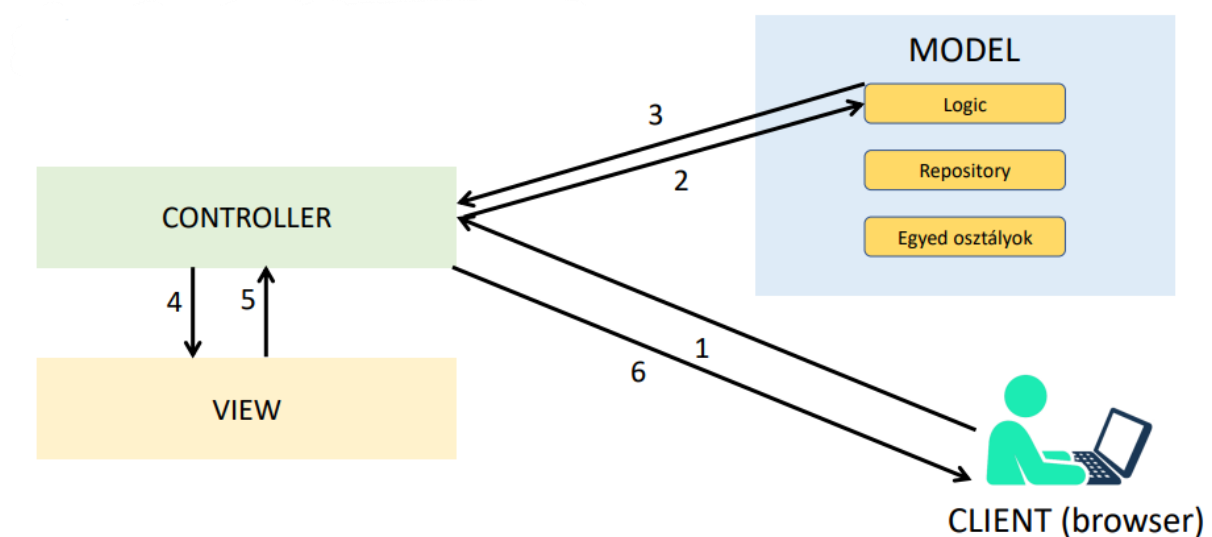
- **Dependency Inversion**
 - A függőségeket ne az őket felhasználó osztály hozza létre.
 - Várjuk kívülről a példányokat interfészeken keresztül.
 - Példány megadására több módszer is lehetséges
 - Dependency Injection
 - Inversion of Control (IoC) container
 - Factory tervezési minta
 - Ha nem követjük, 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

Architekturális tervezési minták

- Nagyobb alkalmazás alapvető osztályait, működési módját és technológiáját határozza meg.
- Három nagyobb architektúra
 - MVC
 - Model-View-Controller
 - MVVM
 - Model-View-ViewModel
 - MVVMC v1
 - Model-View-ViewModel-Controller
 - MVVMC v2
 - Model-View-ViewModel-API-Controller

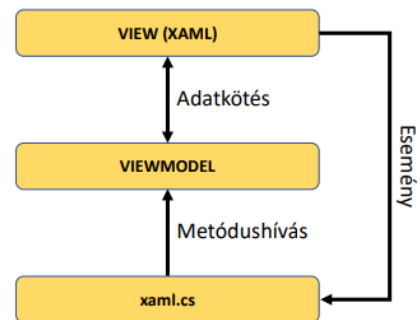
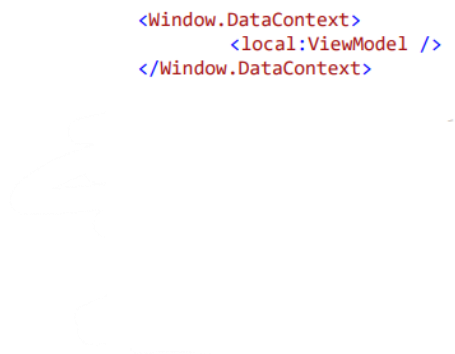
MVC workflow

- A Controller visszaküldi a HTML választ a felhasználó kérésére.
- Nem nyúl már hozzá, a View választ változatlanul visszaküldi.



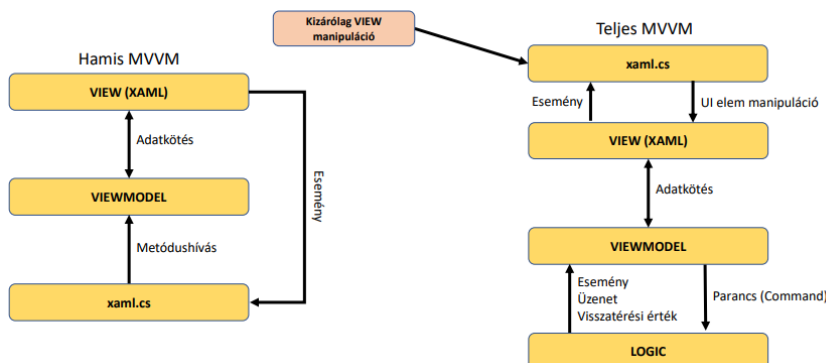
Hamis MVVM

- WPF ViewModel
 - o Egy logic, amit DataContext-nek használunk a Window szintjén.
 - o Gyűjtemények
 - ObservableCollection<T>
 - o Entitások
 - INotifyPropertyChanged megvalósítás
 - o Metódusok
 - Eseményekből hívva
 - o Tulajdonságok
 - UI vezérlőkre kötve
- Még nem MVVM pattern
 - o Logika szétválasztása csak
 - o UI függ a logikától még
 - Az eseménykezelőkben név szerint metódusokat hívunk.



MVVM

- Szabályok
 - o View csak a ViewModel-t ismeri.
 - o View eseményei adatkötéssel vannak kötve a ViewModel valamelyik ICommand típusú tulajdonságához.
 - o ViewModel tudja, hogy adott Commandra melyik Logic metódust kell hívni és milyen paraméterekkel
 - o Logic nem tud a ViewModel-ről
 - o A Logic teljesen átvihető bármilyen nem GUI alkalmazásba is.
- XAML.cs
 - o Csak View segítése
 - o Alapvetően szinte üres



MVVMC v2

- Cél
 - o Konzol kliens kiváltása WPF klienssel
- Vastagkliens fejlesztés
 - o WPF app DLL-ként megkapja az alsóbb rétegeket
 - o Egy eszközön lehet futtatni a teljes stacket
- Vékonykliens fejlesztés
 - o WPF app API-kérésekkel éri el a Controller réteget
 - o A Logic és az alatta lévő rétegek a szerveroldal
 - o A WPF app a kliensoldal
 - o Több WPF kliens is csatlakozhat egyidőben

GOF minták

- GOF - Gang of Four
 - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable ObjectOriented Software
- Jellemzőik
 - o Újrafelhasználás maximális kiszolgálása
 - o Öröklésre vagy kompozícióra építenek

23 db minta		Cél		
		Létrehozási/Creational	Strukturális/Structural	Viselkedési/Behavioral
Hatókör	Osztály	Factory method	Adapter	Interpreter Template method
	Objektum	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Refaktorálás

- Elméletben
 - o Újratervezés/módosítás/előretervezés során azonosítsuk a komponenseket, amiket javítani tudunk.
 - o Válasszuk ki a megfelelő tervezési mintát és alkalmazzuk.
- Gyakorlatban
 - o Fejlesztés közben nem figyelünk a baljós jelekre (code smells).
 - o Átláthatatlan kód és tervezés
 - o Refaktorálunk, majd jobb lesz

Factory method

- Probléma
 - o Az objektumainkat gyakran bonyolult létrehozni és a konstruktor nem elég flexibilis ehhez
- Példa
 - o Létrehozható a Descartes koordinátákkal (X és Y)
 - o Létrehozható Polár koordinátákkal (szög és távolság)
 - o Mind a kettő esetben 2db double értéket várnánk
 - o Nem hozhatunk létre két konstruktort ugyanolyan típusú paraméterekkel
- Megoldás
 - o Egy külső PointFactory osztály, ami segít egy Pointot elkészíteni.

Abstract Factory

- Probléma
 - o Különböző feltételek alapján más és más objektumokat szeretnénk szolgáltatni.
 - o Például
 - Egy stringtől függ, hogy milyen osztályt példányosítunk
 - Egy attribútumtól függ, hogy milyen validációs osztályra van szükségünk.
- Megoldás
 - o Egy ősfactory – sok leszármazott factory
 - o Dictionary vagy reflexió azonosítja a paraméter függvényében a megfelelő factory-t.

Builder

- Probléma
 - o Egy objektum gyártásához rengeteg paraméter kell.
 - o Nagy konstruktorokat szül.
 - o Nem kell gyakran minden paraméter.
 - o Opcionális paraméterek kerülendők
 - o Null értékek kerülendők
- Megoldás
 - o Üres konstruktor
 - o Láncolható gyártófüggvények, amik beállítják a paraméterek egy halmazát.
 - o A gyártás kiszervezése egy különálló osztályba.

Singleton

- Probléma
 - o Jól jönne egy adott helyzetben statikus osztály használata, mert mindig ugyanaz az állapottért kell.
 - o Nem szeretnénk statikus osztályt készíteni, mert nem tesztelhető.
 - o Kellene egy olyan megoldás, hogy egy osztálytól mindig ugyanazt a példányt kérhessük el.
 - o Például: Adatbázis kapcsolattartó osztály (DbContext), konfigurációs beállítások, stb.
- Megoldás
 - o Osztály létrehozása
 - o Egy statikus adattag létrehozása
 - o Abban egy példány létrehozása
 - o Mindig ennek a példánynak a szolgáltatása.

Prototype

- Probléma
 - o Nem szeretnénk mindig objektumokat építeni a nulláról.
 - o Például
 - Tantárgyi követelmények
 - Lemásoljuk a tavalait és néhány apróság változik
 - o Ezt leprogramozni nem triviális, mert alapvetően shallow copy objektumokat kapunk.
 - Jelentése: Referencia típusoknál csak a referenciát kapjuk másolatként
 - Ugyanarra az adatra mutat a két objektum.
 - Egy objektum általában eléggé összetett, van sok érték és sok referencia típus benne.
- Megoldás
 - o Osztályok szintjén Deep Copy metódusok létrehozása
 - o Vagy szerializáció, majd deszerializáció.

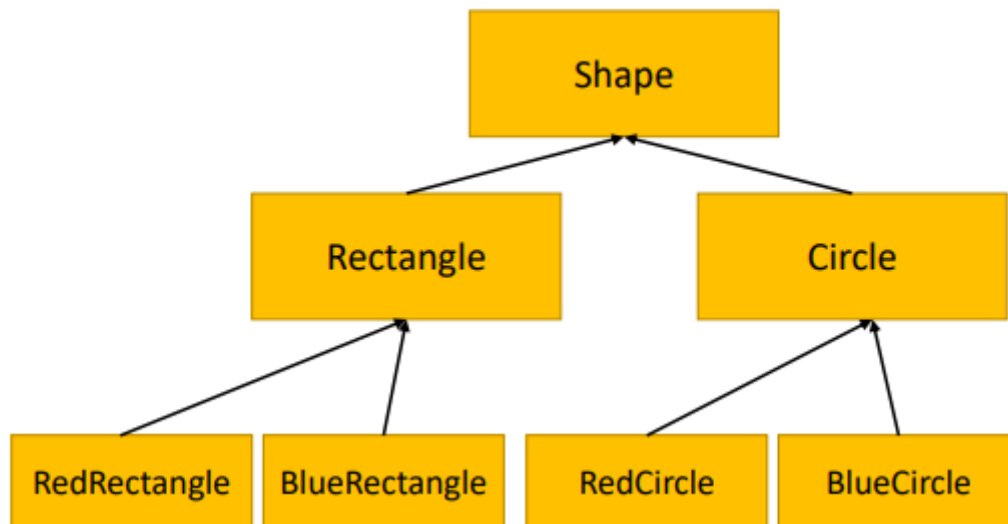
GOF Structural Patterns

Adapter

- Probléma
 - o Össze szeretnénk kötni két rendszert, amelyek nem kompatibilisek.
- Példa
 - o Egy adatmegjelenítő osztály az adathalmazt INamedata interfészen át várja.
 - o Egy adatforrás interfész az adathalmazt INamedata interfészen át biztosítja.
 - o Az adatmegjelenítőnek nem adható át az adatforrás.
- Megoldás
 - o Gyártunk egy NameAdapter osztályt
 - o Ez megvalósítja az INamedata interfészt
 - o Forrásként várja az INamedata interfészt
 - o Megvalósítja a konvertáló logikát.
- Való élet példa
 - o Adapter kábelek

Bridge

- Probléma
 - o Egy osztály két jellemzőtől is függ.
 - o Például alakzatok: szín és forma
- Megoldás
 - o Szét kell bontani az osztályt
 - o A forma osztály várja interfészen keresztül a szín osztályt.
 - o Kompozícióval lehessen összeépíteni őket.



Composite

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

Flyweight

- Probléma
 - o Szeretnénk optimalizálni az alkalmazást, hogy kevesebb memóriát használjon.
 - o CPU kárára tudjuk megtenni.
 - o Nincs konkrét megoldás
 - Sok trükköt biztosít a FlyWeight minta.
- Megoldás
 - o On-the fly propertyk
 - o Objektumok közös részeinek eltárolása egyszer
 - o Újrahasznosított objektumok

Facade

- Probléma
 - o Több osztályt/alrendszer is kell hozzá, amiket körülményes konfigurálni.
- Példa
 - o Egy webes végpont JSON formában adatokat biztosít.
 - o Szükség hozzá egy WebClient, hogy letölthessük a JSON szöveget.
 - o Szükséges hozzá egy JsonConverter, hogy deszerializálhassuk.
- Megoldás
 - o Gyártunk egy SimpleJsonConsumer<T> osztályt.
 - o Legyen képes nagyon könnyen visszaadni a List<T>-t egy URL cím alapján.
 - o Extra logic kódot nem adunk hozzá, csak egyszerűsítük.

Proxy

- Probléma
 - o Adott egy rendszer, amit módosítani nem tudunk (pl.: CRUD subsystem)
 - o Interfészen keresztül érjük el.
 - o Alkalmazás már üzemel.
 - o Többlet kódot szeretnénk beinjektálni, ami a metódushívások előtt lefutnak. (pl.: jogosultságkezelés, telemetria)
- Megoldás
 - o Implementálunk egy új osztályt az elérni kívánt rendszer interfésze alapján.
 - o Az elérni kívánt rendszer ebbe kerül be kompozícióval.
 - o A rendszerrel azonos metódusok lesznek, amik meghívják a rendszer ugyanilyen nevű metódusait.
 - o Meghívás előtt futtathatunk többlet kódokat
 - Jogosultságkezelés
 - protection proxy
 - Helyi metódus egy távoli metódust hív API-n keresztül
 - remote proxy

Decorator

- Probléma
 - o Szeretnénk egy objektumot ellátni többlet jellemzőkkel.
- Példa
 - o Egy képet el szeretnénk látni kerettel
 - Ő maga is legyen egy kép.
 - o Egy hallgatót el szeretnénk látni Erasmus adatokkal
 - Utána ugyanolyan hallgatóként tudjuk használni.
- Megoldás
 - o Leszármazott osztály készítése vagy közös interfész megvalósítása.
 - o Kompozícióval a kibővítendő objektum eltárolása.
 - o Többlet infók hozzáadása.
 - o Új objektum használata gond nélkül.

GOF Behavioral Patterns

Iterator

- Probléma
 - o Legyen bármilyen gyűjteményünk ezeket szeretnénk egy bejárható interfészen keresztül elérni.
- IEnumerable<T> interfészen át érjük el a gyűjteményeinket.
- Megoldás
 - o Adatszerkezeten implementáljuk az IEnumerable<T> és egy külső bejáró osztályon az IEnumerator<T> interfészt.
 - o Az IEnumerator<T> előírja az alábbi metódusokat:
 - void Reset()
 - Gyűjtemény elejére visszaállítás
 - bool MoveNext()
 - Következő elemre lépés
 - T Current
 - visszaadja az aktuális elemet
 - o yield return

Chain of responsibility I.

- Probléma
 - o Egy olyan folyamatot implementálunk, ahol egy objektum sok részfeladaton megy keresztül
 - o Példa
 - 10 db validációt egymás után meghívunk rajta és ha bármelyik hibára fut, akkor nem nézzük a többit (feltételes továbbpasszolás)
 - 10 db naplózó metódusnak egymás után átadjuk az objektumot és mindenképp mindegyiken végigmegy (feltétel nélküli továbbpasszolás)
 - o Első esetben egy nagyon összetett elágazás jön létre.
 - o Példa alapján 10 db metódushívás egymás alatt.
- Megoldás
 - o Validáló/logoló osztályokból készítünk egy gyűjteményt
 - o Végigmegyünk az elemein és mindegyiknek átadjuk az objektumot.

Chain of responsibility II.

- Probléma
 - o Egy olyan folyamatot implementálunk, ahol egy fizetési objektum egy döntési hierarchián fut végig.
 - o Csak a közvetlen feletteshez intézünk kérelmet, láthatatlan számunkra, hogy ő maga saját hatáskörben jóváhagyta, vagy pedig a saját feletteséhez fordult.
 - Kapunk egy választ, hogy igen/nem.
- Megoldás
 - o Boss osztály
 - o Példányok tárolják a saját pénzügyi hatókörüket.
 - o Példányok tárolják a saját felettesüket.
 - o Példányok döntést hoznak vagy továbbítják a kérelmet felfele.

Visitor

- Probléma
 - Hívó és hívott szétválasztása.
 - Hívó (Logic) tudhat a hívottról (subject), de fordítva tilos.
 - A hívott dönthessen róla, hogy vele éppen lehet-e dolgozni.
 - Működjön az egész gyűjteményekkel is (több hívó és több hívott)
- Megoldás
 - Interfészeken át érik el egymást
 - Hívottnak legyen Accept() metódusa
 - Hívónak legyen Visit() metódusa
 - A hívott az Accept() metódusában döntést hoz és egyben meghívja a hívó Visit() metódusát.

Observer

- Probléma
 - A subject állapotváltozásairól szeretnénk különböző feliratkozókat értesíteni.
 - De nem kéne tudnunk róla, hogy milyen feliratkozók vannak és hogy vannak-e.
 - Példa
 - Változott egy személy fizetése, így minden felületen, adattárolóban módosítsuk.
- Megoldás
 - Feliratkozó osztályok valósítanak meg egy ISubscriber interfészt.
 - Írjon elő egy StateChange() metódust
 - A subject kezelje a feliratkozókat Subscribe(), UnSubscribe()
 - Állapotváltozáskor hívja meg az összes feliratkozó StateChange() metódusát.
 - A feliratkozók tegyék meg a frissítési lépéseket.

Command

- Probléma
 - Egyirányú függés van két nagy réteg között.
 - UI ismeri a Logic-ot.
 - Ha lecseréljük egy másik Repository-ra, akkor nagyon sok helyen a UI kódját át kell írunk.
- Megoldás
 - Közvetve ismerje csak a UI a Repository-t.
 - A közvetítő osztályban ne kelljen átírni bármit, ha módosul a Repository.
 - A közvetítő osztályba pedig extra kód is legyen írható.

Mediator

- Probléma
 - Egyirányú függés van két nagy réteg között.
 - Ne legyen semmilyen irányú függés.
 - Egy közvetítő osztályon keresztül lehessen csak beszélgetni két osztálynak.
- Megoldás
 - Legyen egy Mediator osztályunk.
 - Lehessen regisztrálni egy üzenetfolyamára.
 - Lehessen beküldeni egy üzenetet valamelyik üzenetfolyamára.

Strategy

- Probléma
 - o Szeretnénk különböző titkosításokat használni.
 - o Szeretnénk a logikában titkosítva levelet küldeni.
 - o Szeretnénk a programunkhoz a jövőben új titkosítási módszereket adni flexibilisen.
- Megoldás
 - o Ős referenciát használjuk mindenhol.
 - o Leszármazottakkal felül lehessen definiálni az absztrakt titkosítás metódusát az ősnek.

Template method

- Probléma
 - o Szeretnénk bizonyos folyamatok egyes részeit a későbbiekben módosítani.
 - o Példa
 - PersonLogic képes szűréseket végezni embereken.
 - Leszármazottakkal lehessen felüldefiniálni a beolvasást és validálást.
- Megoldás
 - o A logika meghívja a saját virtuális Import() és Validate() metódusait.
 - o Ezeket leszármazottal felül lehet írni, ha szeretnénk.

Memento

- Probléma
 - o Undo funkciót szeretnénk implementálni egy entitásra.
 - o Kívülről ne is nagyon lássuk, hogyan oldja meg.
 - o Legyen képes bármikor visszaállni az előző állapotába.
 - Akár legyen képes bármely előző állapotra visszaállni.
- Megoldás
 - o Memento<T> osztály készítése
 - o Az entitás kompozícióval eltárolja a Memento<T> osztályt (ami egy state store)
 - o Felhasználjuk az entitásban a prototype mintát.
 - o A deep copy-t elmentjük a Memento-ba.
 - o Memento biztosítson Undo() metódust.

State

- Probléma
 - Különböző állapotváltozásokat szeretnénk egy objektumban tárolni.
 - Példa - Hibajegy
 - Inicializálás
 - Created state
 - Send()
 - Send state
 - Close()
 - Closed state
 - State-machine diagrammal tervezzük meg.
- Megoldás
 - Osztályba ITicketState interfészeket át be vesszük az állapotot.
 - Létrehozunk a Send() és Close() metódusokat.
 - Az ITicketState-ben is van Send() és Close() metódus
 - Minden állapotnak egy-egy implementációt készítünk.

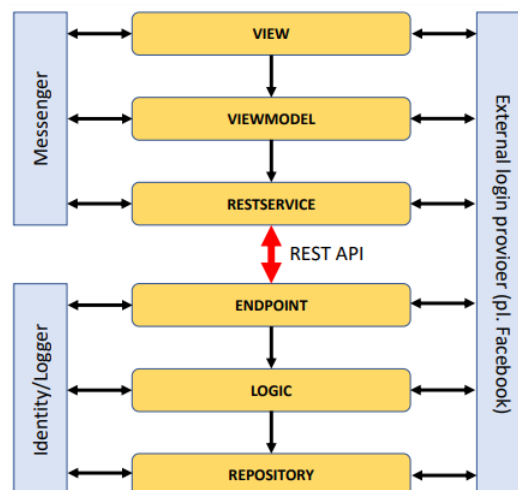
Interpreter

- 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
 - Elkészítjük az írásjeleket reprezentáló osztályokat (Token)
 - Lexer elkészítése
 - Parser elkészítése

Domain alapú tervezés és mikroszolgáltatások

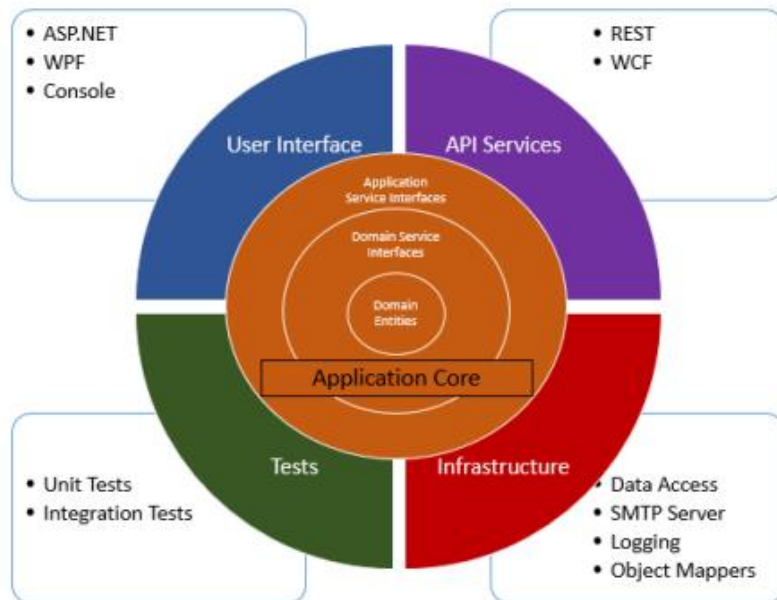
Aspect-ek

- Olyan komponensek, amiket minden réteg használ.
- Tipikus elvárások az aspect-ekkel szemben
 - Ne kövessenek el rétegsértést.
 - Legyenek szűk funkcionalitásúak.



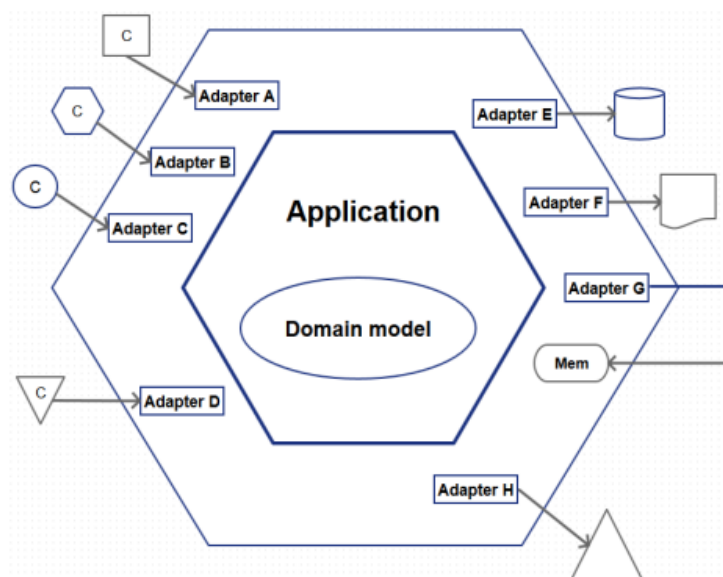
Onion architecture

- Application core = Logic
- Logic felett több réteg is elhelyezkedik.
- Mindegyik más-más célt szolgál ki.



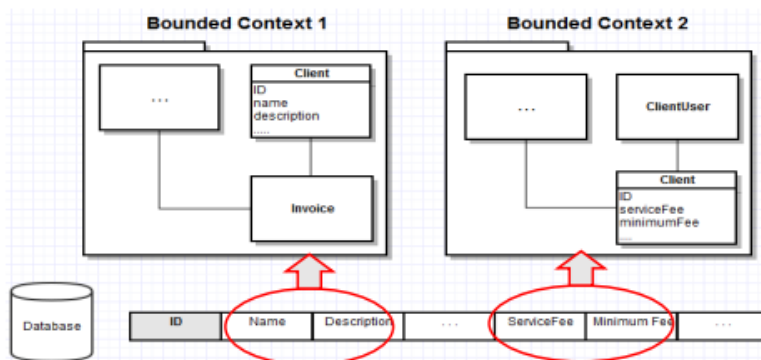
Hexagon architecture

- Application core = Logic
- Logichoz külső rendszereket csatolunk adaptereken keresztül.
- Külső rendszerek
 - o Adattárolás
 - o API végpontok
 - o Webes UI
 - o Logolás
 - o Stb...



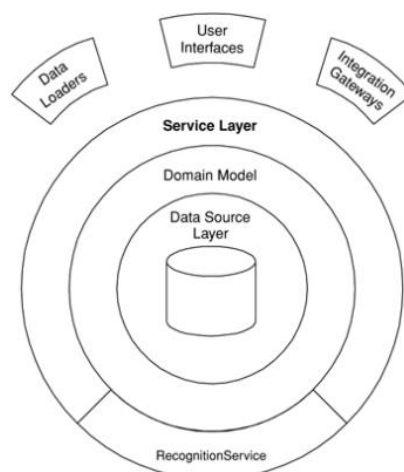
Domain-driven design

- Lényege
 - o A rétegzés ne attól függjön, hogy MVC vagy API vagy bármilyen a UI elérési technika.
 - Attól függjön a rétegzés, hogy mit akarok csinálni az adattal.
- Nem az adatbázissal kezdjük a modellezést, hanem a funkciókkal.
- Bounded Context-eket hozunk létre
 - o Domain model lesz belőlük
 - o Jelentése: Egy User tábla szerepelhet a Szállítás domain modelben és a Számlázás domain modelben is.
 - o DRY elveknek ellentmond, de csak látszólag, mert a Data Mapper / ORM majd valójában ugyanarra az 1 db táblára mappeli le.
 - o Hibalehetőségek
 - Bloated domain objects (túl sok felelőség)
 - Anemic domain objects (túl kevés felelőség)



Service Layer

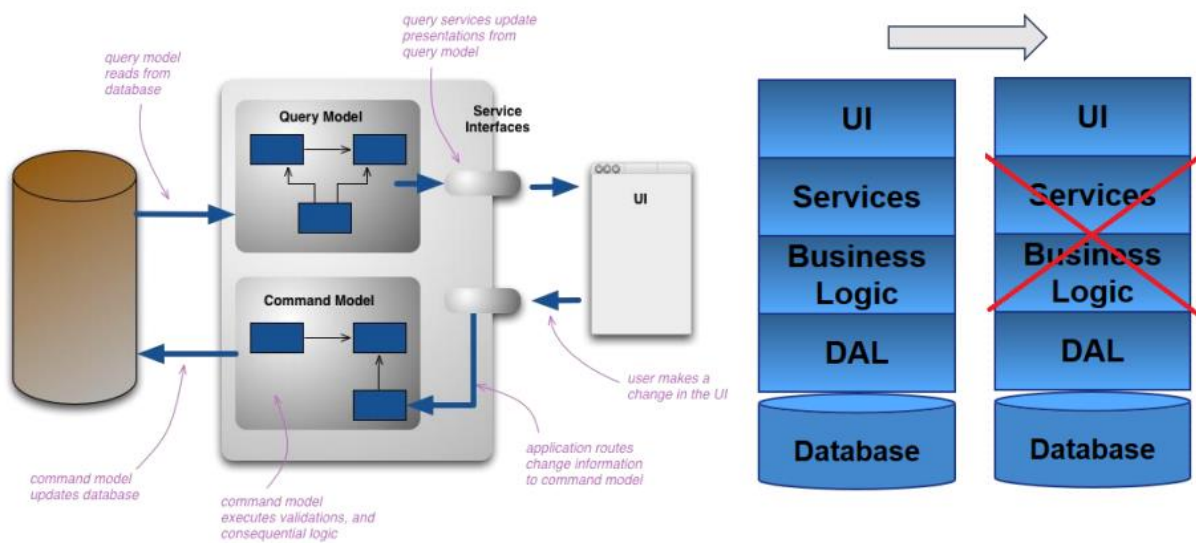
- Domain model egységbezárrja a Domain Entityket és azok üzleti műveleteit.
- SOA filozófiát valósít meg (Service-Oriented Architecture)
 - o Microservices
- Feladata
 - o Hívások fogadása, továbbítása a Domain Logic felé.
- Feladata lehet még a tranzakciókezelés és a lock is.
- Alsóbb rétegekben megjelenik ettől függetlenül az adatbázis szintű tranzakciókezelés is.
- UI csak egy service a sokból.



Írás vs olvasás

- Nagy rendszerek esetén általában SOK olvasási művelet és KEVÉS írási művelet történik.
- Írási műveletek
 - o Tipikusan egy bounded context-be akarunk írni.
 - o Kell minden alrendszer hozzá.
- Olvasási műveletek
 - o Dashboard és Reports funkcionalitás nagyon gyakori.
 - o Általában több bounded context-ből kell összeszedni az adatokat.
 - o Egy csomó alrendszer kikerülhető akár.
- Ez szétválasztható lenne két nagy alrendszerre
 - o CQRS (Command-Query Responsibility Segregation)

CQRS



Olvasás gyorsítása

- A User lásson mindig régi adatot.
- Megoldás
 - o Persistent View Model
 - Minden éjfélkor például egy adatbázisba mentjük ki az összes user dashboardját.

CQRS hibalehetőségek

- Query oldal – Olvasási műveletek
 - o Egyedi keresések problémája
 - Generálható minden éjjel a top 10 népszerű keresés találati oldala.
 - A ritka keresések majd tovább tartanak.
 - Adatbázisok optimalizálhatóak keresésekre.
 - Gyors keresésre optimalizált DB: Elasticsearch
- Command oldal – Írási műveletek
 - o Hibára futás ritka
 - o Szinkron hibajelzés feleslegesen lassít
 - o Aszinkron hibajelzés
 - Sikeres foglалás
 - Ha baj van, akkor email küldése, hogy hiba történt.
 - Aszinkron reagáló mechanizmusok

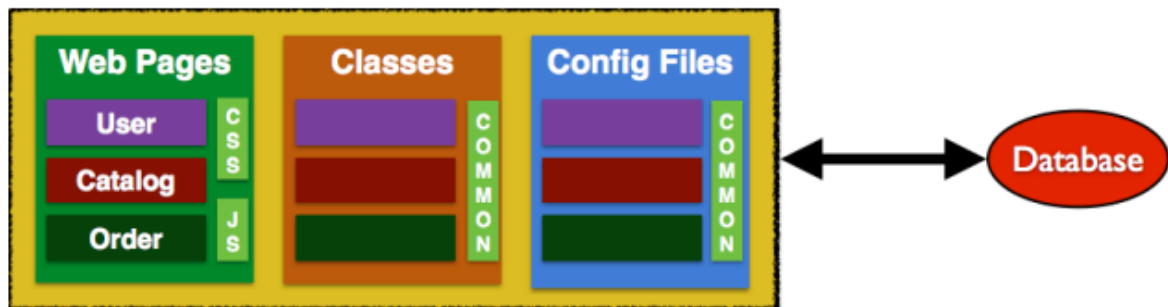
Event Sourcing

- 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.
 - o Feltorlódnak a kérések és használhatatlan lesz a rendszer.
- Megoldás
 - o Várósorba mentés – Event Store
 - o Technika
 - Redis
 - RabbitMQ
 - MQTT
 - HiveMQ
 - o Ezek az adatbázisok arra vannak optimalizálva, hogy villámgyorsan képesek legyenek elmenteni kéréseket, nagyságrendekkel gyorsabban, mint egy relációs adatbázis.

Event Sourcing + CQRS + DDD

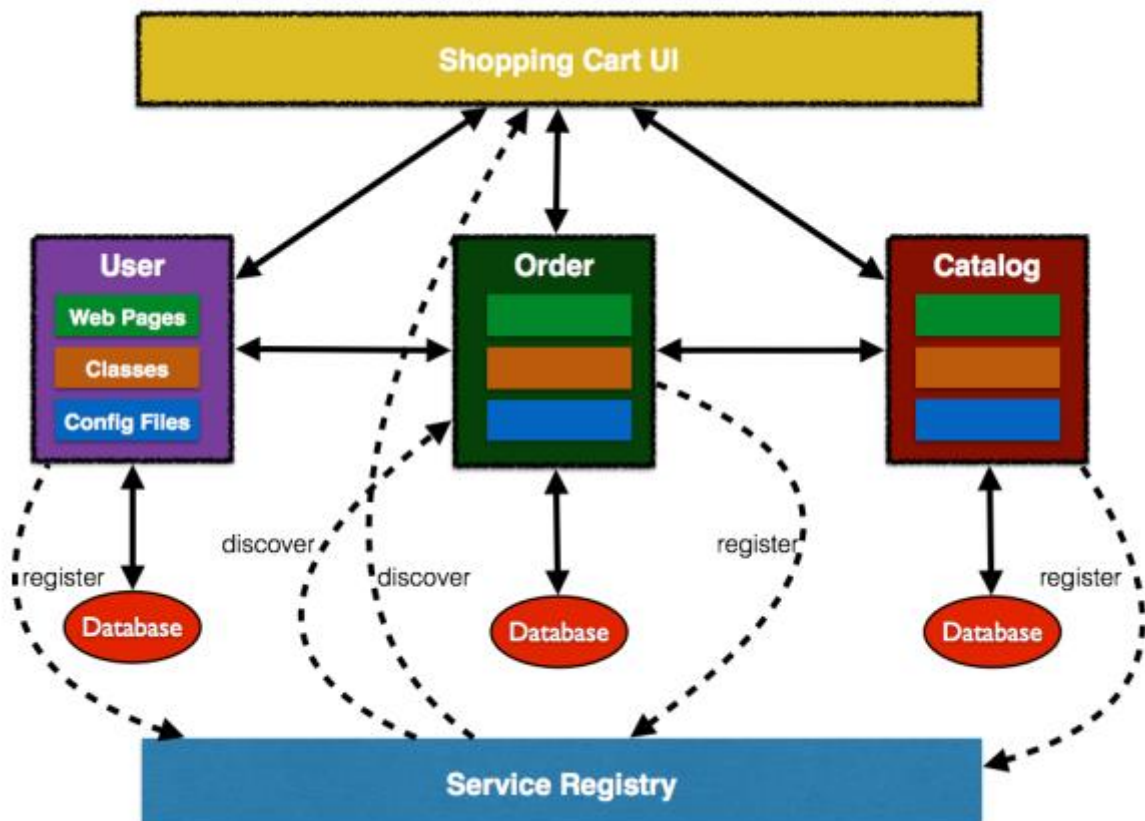
- Előnyei
 - o Nagy teljesítmény
 - o Egyszerűbb a rendszerek összeépítése
 - o Könnyű hibakeresés, tesztelés
 - o Event Store-ból extra üzleti adat is kinyerhető.
- Hátrányai
 - o Reporting bonyolult
 - o Nagyobb tárigény
 - Persistent View Model eltárolása
 - o Hibás kérések visszajelzése nem azonnali.

Monolitikus alkalmazás



Mikroszolgáltatások

- Jellemzői
 - o Önálló alkalmazások
 - o Felbontás alapja
 - domain model → bounded context
 - o Önálló fejlesztési ciklus
 - o Önállóan tesztelhetőek (service stubokkal)
 - o Önállóan skálázhatóak
 - o Akár más-más nyelven írhatóak
 - o Egymással valamilyen közösen ismert protokollon keresztül beszélgetnek.
 - o Tipikusan konténerbe zárunk egy-egy mikroszolgáltatást.



Mikroszolgáltatás felépítése

- Belső működés
 - o Hagyományos rétegezéssel épül fel.
 - o Saját adatbázissal rendelkezik/rendelkezhet.
 - o Kommunikáció
 - REST API vagy Message bus protokollok
- Külső elérés
 - o Tipikusan REST API-n keresztül
 - o UI is egy mikroszolgáltatás, ami megjelenítésért felelős.

Mikroszolgáltatás kommunikáció

- Mediator technológia
 - o AMQP – Advanced Message Queueing Protocol
 - Általános, nyílt protokoll
 - Tipikusan PC/WEB
 - o MQTT – Message Queue Telemetry Transport
 - ISO szabvány
 - Publish – Subscribe üzenetküldésre tökéletes
 - Minimális overhead
 - Tipikusan mobil/IoT
 - Brokers