

SZOFTVERTECHNOLÓGIA JEGYZET

Fehér Ágnes
Végvári Vilmos

Tartalom

Software engineering.....	4
Unified Modeling Language	4
Use Case Diagram	4
Activity Diagram.....	5
Component Diagram.....	6
Deployment diagram	7
Class Diagram.....	7
Objektum orientáltság	9
Package Diagram.....	9
Object Diagram	10
Sequence Diagram	10
Communication Diagram	11
Interaction Overview Diagram.....	11
State Machine Diagram.....	12
Timing Diagram.....	13
Composite Structure Diagram.....	14
Profile Diagram	14
Összefoglalás.....	15
Objektumorientált tervezési elvek	16
A SOLID alapelvek	16
Single Responsibility Principle (SRP)	16
Open/Closed Principle(OCP)	16
Liskov Substitution Principle (LSP)	17
Interface Segregation Principle (ISP).....	17
Dependency Inversion Principle (DIP).....	18
További elvek:	18
Don't Repeat Yourself (DRY)	18
Single Choice Principle (SCP).....	18
Tell, dont't ask (TDA).....	18
Law of Demeter (LoD)	18
Design Pattern.....	18
Bevezetés.....	19
Definíciók	19
Problémák.....	19
Feladatmegoldás folyamata.....	20
Mitől „más” a szoftver?	20
A szoftverfejlesztés	20
A szoftverfejlesztés modellezése	21

Folyamatmodell:	21
Modellezések lehetőségei.....	22
Életciklus modell.....	23
Szoftver életciklus	23
Szekvenciális életciklus modellek.....	23
Vízesés modell.....	23
V modell	23
Inkrementális modellek	24
Iteratív modellek.....	24
Extreme Programming	24
Scrum	24
Szoftver életciklus modellek	25
Hagyományos (vízesés) VS agilis megközelítés.....	25
Agilis szoftverfejlesztés	25
Lean.....	26
Folyamatfejlesztési modellek	27
Érettség	27
Képességi-érettségi modellek.....	27
CMM – Lépcsős modell (kevés).....	27
SPICE – folytonos modell	28
Felmérés.....	28
CMMI – Integrált modell (most v1.3).....	29
CMMI folyamatok:	29
Folyamatok, GG.....	30
Egyéb folyamatfejlesztési modellek.....	31
PSP.....	31
CMMI és PSP	31
TSP.....	32
IDEAL.....	32
Agilis folyamatfejlesztés.....	32
Követelmények kezelése.....	33
Követelmény fejlesztés (RE – Requirements engineering)	33
Követelmények típusai.....	35
Követelmények modellezése	35
Követelmények agilis környezetben	37
Tervezés, implementáció	38
Tervezés / Design	38
Definíciók	38
A tervezés folyamata.....	38

Szoftvertervezési alapelvek és fontos elemek	38
Architektúrák, döntések, tervezési minták	38
Felhasználói interfész (UI) tervezése	39
Szoftvertervezés a CMMI és az Automotive SPICE modellekben	39
Design dokumentálása	40
Implementáció / kódolás	40
Kódolási szabványok alkalmazása	40
Becslés a szoftvertervezés során	40
Tervezés és implementáció agilis környezetben	41
Agilis Design elemek:	41
Az agilis design és projekttervezés	41
Agilis fejlesztési technikák:	42
Agilis fejlesztő	42
Tesztelés	43
Miért szükséges tesztelni?	43
A tesztelés definíciója	43
A szoftverfejlesztés 7 alapelve	43
A tesztelési folyamat elemei	43
Tesztelés a CMMI-ben	44
Tesztelés dokumentálása	44
Hibák	45
Teszt lezárása	45
Tesztek típusai	46
• Statikus tesztelés	47
• Dinamikus tesztelés	48
A tesztelés hatékonysága	50
Tesztelés agilis környezetben	50
A szoftverprojektek menedzsmentje	51
Projektmenedzsment (PM)	51
Projektirányítási módszertanok	51
CMMI-ben	52
PM és szoftverfejlesztés	53
Projektmenedzsment agilis környezetben	56
Támogató folyamatok	57
Konfigurációmenedzsment (CM)	57
Verziókezelés (VC)	57
Kockázatmenedzsment	58
Minőségmenedzsment	58
Mérés és elemzés	59

Szoftvertechnológia jegyzet

Software engineering

Szoftver fejlesztés folyamata:

- Követelmények definiálása és specifikáció: kitalálni, hogy mit akar a felhasználó
- Tervezés: a szoftver architektúrájának, elemeinek, és viselkedésének megtervezése
- Kódolás, megvalósítás: a szoftver megírása
- Tesztelés: szisztematikus felderítése és javítása a hibáknak
- Átadás: a program telepítése a végfelhasználóknál, és a felhasználók oktatása
- Karbantartás: a szoftver futtatása és hibák javítása

NEM lépések! Egész életében végig követik a szoftvert.

Unified Modeling Language

Modellezés

A modell a valóság egyszerűsített képe.

Egy jó modell a fontos részletekre koncentrál, és nem foglalkozik a feleslegesekkel.

Nézetek

Néha ugyanazt a modellt több célra is használják és több fajta cél közönség nézi, ezért létrehozhatunk különböző nézeteket. Mindegyik nézet a csak az adott célközönség számára lényeges részleteket mutatja.

UML

Meghatározza a szintaxist (hogy néznek ki a diagram elemei) és a szemantikát (a diagramon bemutatott modell mit jelent).

A szemantikának a különböző diagramokon konzisztensnek kell lennie.

Strukturális/statikus szemantika: Bemutat egy elemet, egy adott időpontban. (Component, Deployment, Class, Package, Object, Composite state, Profile diagram)

Viselkedési/dinamikus szemantika: Azt mutatja meg, hogyan változik az idő során egy elem. (Use case, Activity, Sequence, Communication, State, Timing, Interaction overview diagram)

Use Case Diagram

A rendszer funkcionális követelményeit határozza meg.

Mit kéne a rendszernek csinálni, és hogyan használják a rendszert.

Modellezi a felhasználókat és a rendszer hatáskörét.

A rendszer belső szerkezetét nem látni, csak a főbb tevékenységeit, és a kapcsolatát a külvilággal.

Elemei:

Actor: Olyan szerepek, amit a felhasználók

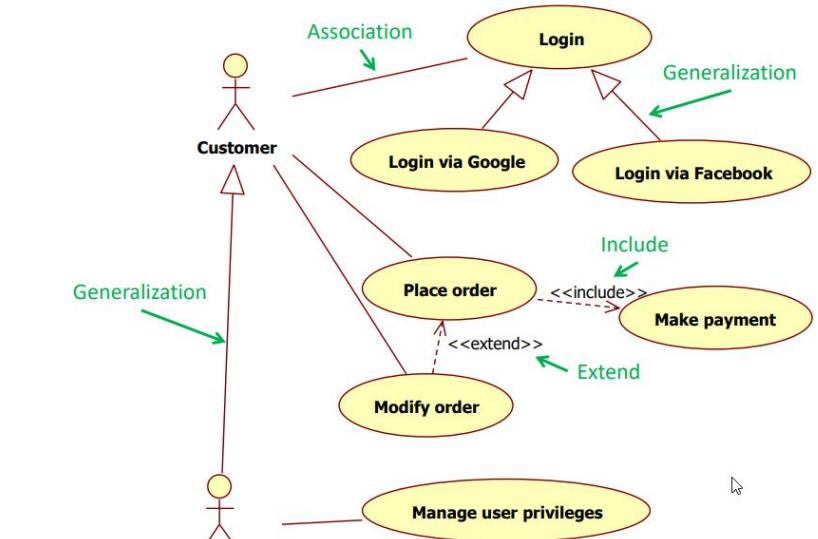
vehetnek fel a rendszerrel való interakciókor.

Use Case: Egy párbeszédet képvisel a rendszer és a felhasználók között, a felhasználó szemszögéből.

Műveletek és interakciók specifikus sorozata, az actorok és a rendszer között.

Kapcsolatok:

- **Association:** Actor és use case között. Összeköti a use case-t a az actor-ral aivel kapcsolatba kerül.
- **Generalization:** Actor-ok közötti „is a” kapcsolat. Az egyik actor megtudja csinálni az összes use case-t amit a másik tud.
- **Generalization:** Use case-ek közötti „is a” kapcsolat. Az egyik use case a másik use case viselkedését specifikusabbá teszi.
- **<<extend>> kapcsolat:** Az extending use case extra viselkedéssel bővíti az egy vagy több extended use case-t az extension pontjaikon.
- **<<include>> kapcsolat:** Az included use case valamikor bekerül majd az including use case-be.



Dr. Balázs Simon, BME, IIT

System boundary box (opcionális): Egy olyan doboz, ami rendszerhatárt szab a use case-eknek. minden use case ami rajta kívül van az a rendszer hatáskörén kívül van.

Package(opcionális): Különböző elemeket lehet csoportosítani vele.

Activity Diagram

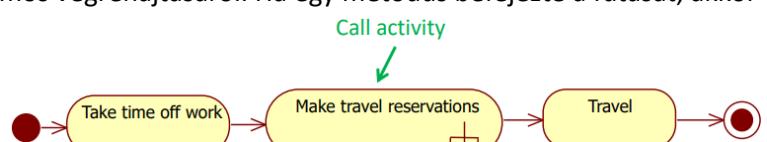
Az activity diagramok grafikus reprezentációi a fokozatos tevékenységek munkafolyamatainak és olyan tevékenységeknek munkafolyamatainak amik támogatják a választást, iterációt vagy konkurenciát.

Általában egy darab use case esetleg egy felhasználó és a rendszer közötti folyamat vagy egy algoritmus lépéseinél modellezésére használják.

Egy hivatalos módja a funkcionális követelmények leírásának.

Speciális állapotgép a metódusok/operációk párhuzamos végrehajtásáról. Ha egy metódus befejezte a futását, akkor lép a következő állapotba.

Activity: A folyamat különböző lépéseinél modellezésére. Egymásba lehet ágyazni őket. Call activity: Meghív egy másik activity-t.



Initial node: Az activity kezdőpontja. Lehet több is, olyankor egyszerre indulnak el, és több folyamat fut egyszerre.

Control flow: Irányított kapcsolat két activity között (forrás és cél). Akkor kezdődik a cél activity, ha a forrás befejeződött.

Final node: Ahol a végrehajtási folyamat végetér.

Activity final node: minden folyamat végetér.

Flow final node: Csak az adott folyamat ér véget, a többi folytatódik.

Guard condition/Őrfeltétel: A control flow-hoz kapcsolt feltétel.

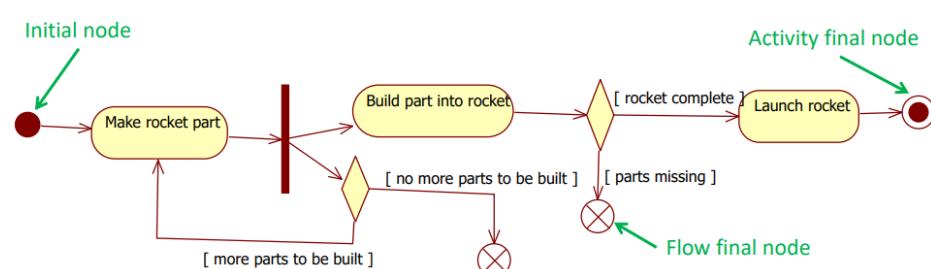
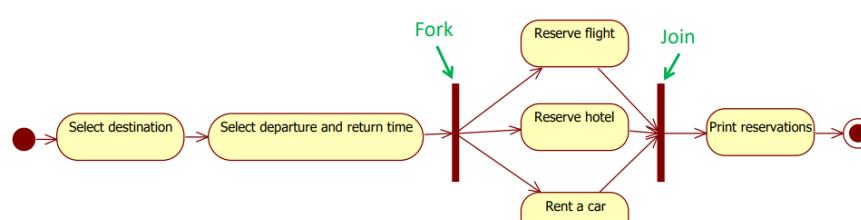
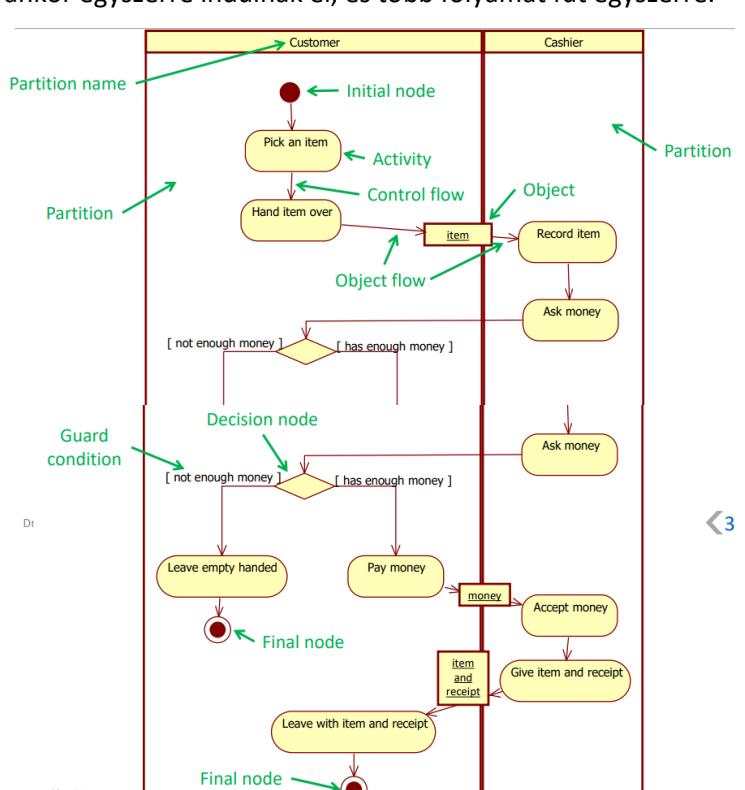
Partition: Ezen belüli activity-ket a hozzá tartozó felhasználó vagy rendszer rész végzi el. Lehet hierarchikus.

Object: Adatokat reprezentál, amiket az activity-k adhatnak egymásnak.

Object flow: Adatokat szállít az activity-k között.

Decision: Őrfeltétel alapján dönti el, hogy hogyan folytatódik a folyamat. Minimum egy folyamat kiválasztódik és végrehajtódik.

Fork: A folyamatot több egyszerre zajló folyamattá bontja.



Signals:

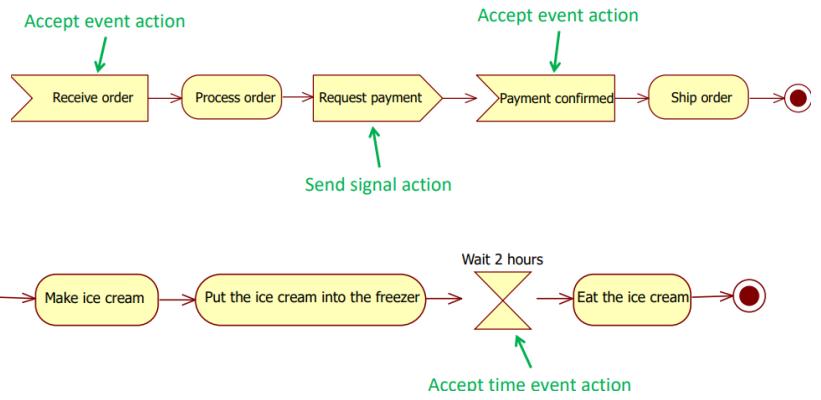
Activity-ket események bekövetkezései (signals) is elindíthatják.

Accept event action: Vár egy eseményre/jelre, és egy új folyamatot kezd.

Send signal action: Jelet küld.

Time events:

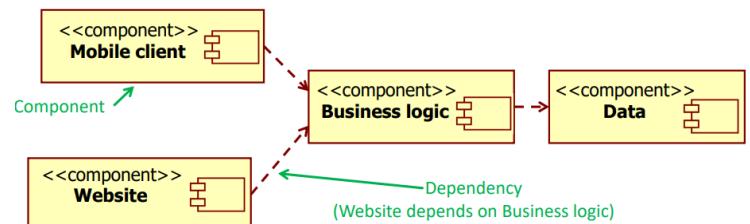
Accept time event action: Elindít egy folyamatot egy bizonyos időpontban.



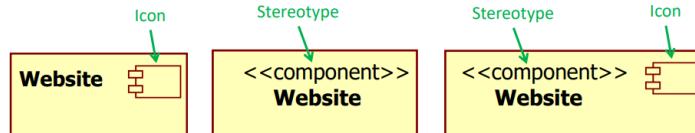
Component Diagram

A diagram egy rendszer komponenseit és az azok közötti kapcsolatot mutatja be.

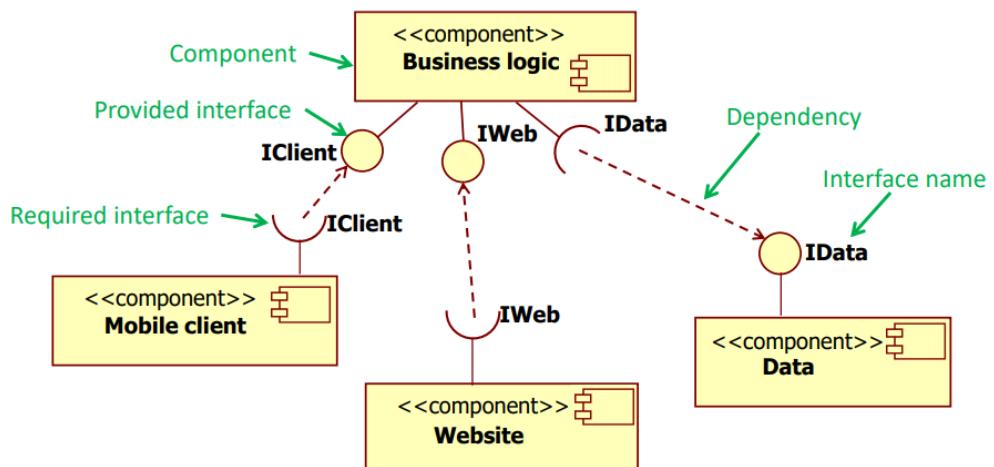
Software component: Olyan egység, amiben meghatározott interfészek és függőségek vannak.
Dependency: Ha „A” függ „B”-től az azt jelenti, hogy ha „B” megváltozik, akkor a változás érintheti „A”-t is. A függőség fajtája nincs megadva.



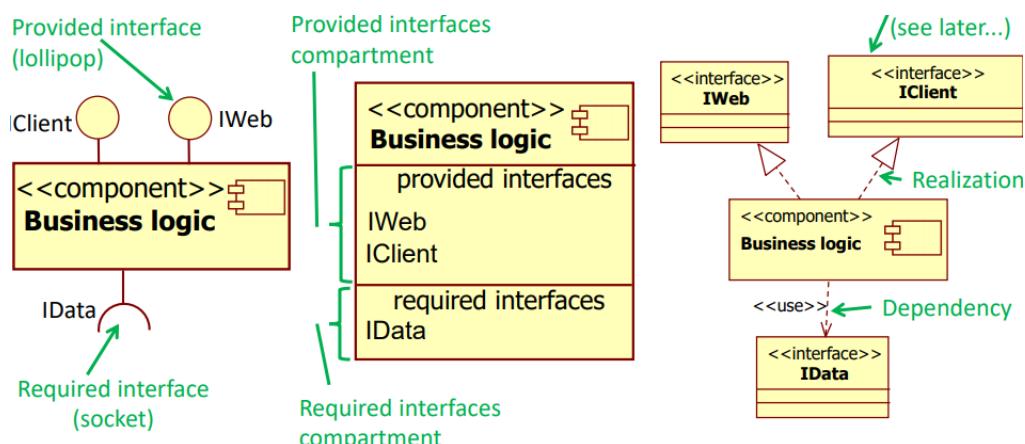
Komponens jelölések:



A diagram bemutathatja a komponensek megadott és szükséges interfészeit is.



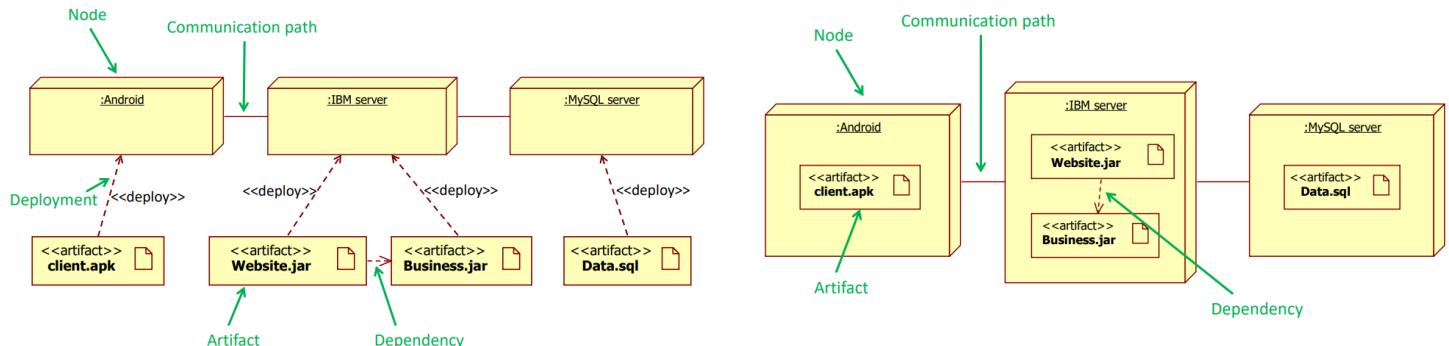
Interfész lehetséges jelölései:



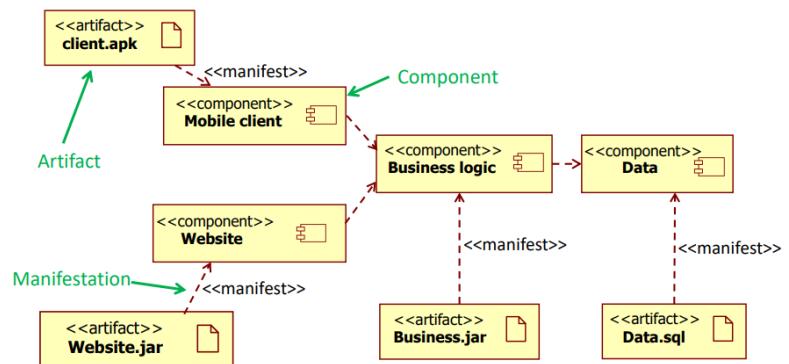
Deployment diagram

Olyan konstrukciókat ír le, amik képesek bemutatni a végrehajtási architektúráját egy rendszernek. A hardware és software topológiát is modellezheti.

Node: Virtuális (logikai) számítógépes erőforrás, amire az Artifact-okat telepíteni lehet. Két node közötti kommunikáció asszociációval valósítható meg.



Artifact: Információkat jelképez. Egy komponens fizikai, testet öltött formája.



Class Diagram

Szoftver architektúra dokumentálására.

Részei (classifiers):

Class: Objektum orientált osztály. Objectumok csoportja, amiknek azonos a viselkedésük. Metódusok, attribútumok.

Interface: Egy csoportnyi metódus. Implementálhatja egy osztályt.

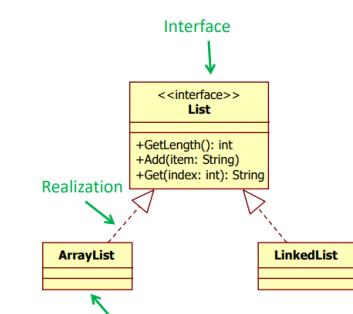
Láthatóságok: private(-), public(+), protected(#), package(~) (in Java protected implies package visibility)

Statikus attribútum: az osztály összes előfordulásánál azonos. (Class diagrammon alá van húzva)

Statikus metódus: Nincs this pointere. Nem lehet override-olni. (Class diagrammon alá van húzva)

Kapcsolatok:

Realization: Osztályok vagy komponensek implementálhatják az interfészket.



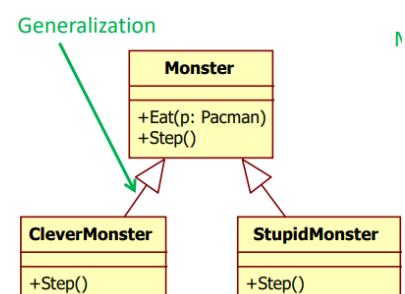
Generalization: Öröklődés interfészek vagy osztályok között.

„is a” kapcsolat. Bővíti az ős viselkedését.

Virtuális metódusok: A származtatott osztály felül írhatja őket.

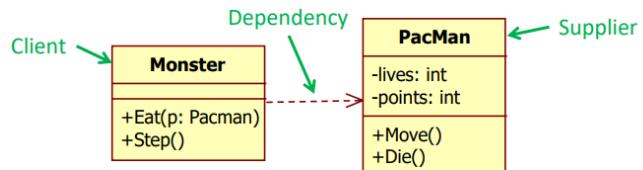
Absztrakt metódus: Virtuális metódus, implementáció nélkül. Egy nem absztrakt származtatott osztálynak kell implementálni. (Class diagrammon dőlt betűvel)

Absztrakt osztály: Nem lehet példányosítani. Általában van minimum 1 absztrakt metódusa, bár nem kötelező. (Class diagrammon dőlt betűvel)

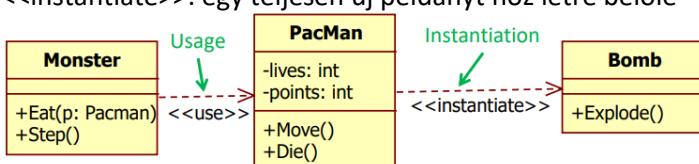


Dependency: Ha „A” függ „B”-től az azt jelenti, hogy ha „B” megváltozik, akkor a változás érintheti „A”-t is. Egy gyenge, ideiglenes kapcsolat. Egy metódushívásig tart.

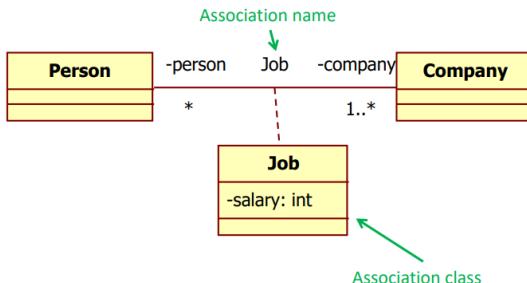
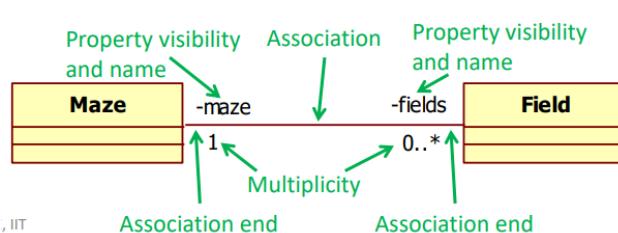
<<use>>: szüksége van rá a teljes implementációjához
<<instantiate>>: egy teljesen új példányt hoz létre belőle



The Monster uses the PacMan class:
it calls the Die() method of PacMan when he Eats the PacMan

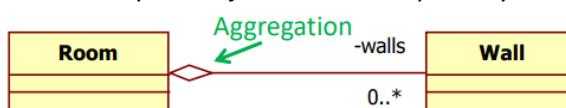


Association: Egy erős, tartós kapcsolat. Az egyik osztályban van egy attribútum ami a másik osztály egy példánya.



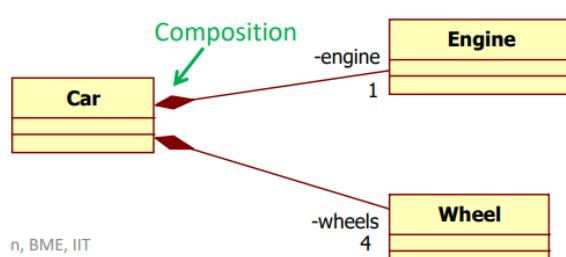
Aggregation, Composition: Ha az egyik összecsoporthoz a másik több példányát.

Shared aggregation: gyenge aggregáció, ha az egyik rész object más aggregációban is részt vehet.



A room has walls, but a wall may be shared between rooms

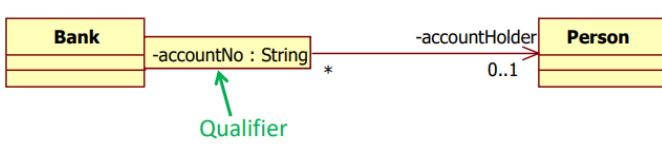
Composite aggregation: erős aggregáció, maximum 1-ben lehet csak a rész objektum.



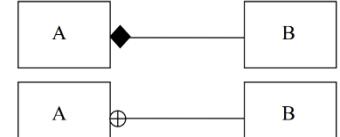
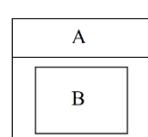
A car contains an engine and four wheels.

If the car is destroyed,
the engine and the wheels
are also destroyed.

32



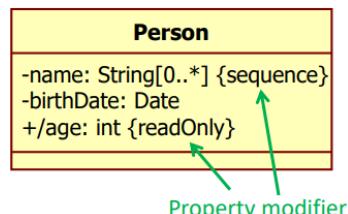
Qualifier: Külön objektumokra lehet jelölni vele a referenciákat.



Nested class: Beágyazott osztály, egy másik osztályon belül van deklarálva.

Property modifiers: Specifikusabbá teszik a property-k jelentését.

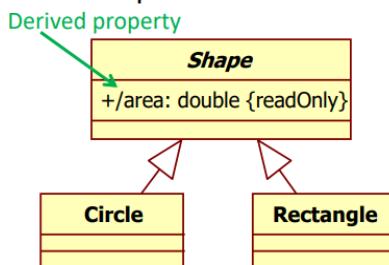
Modifier	Meaning
readOnly	the property is read only
union	the property is a derived union of its subsets
subsets <propname>	the property is a proper subset of the property called <propname>
redefines <propname>	the property redefines an inherited property called <propname>
ordered	values are sequentially ordered (i.e. keeps insertion order)
unordered	values are sequentially not ordered (this is the default) (i.e. may not keep insertion order)
unique	there are no duplicates in a multi-valued property (this is the default)
nonunique	there may be duplicates in a multi-valued property
sequence (or seq)	the property represents an ordered bag (nonunique and ordered)
id	the property is part of the identifier for the class



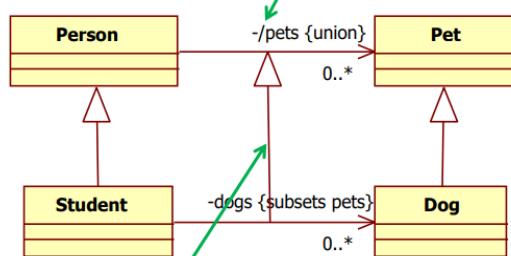
Property modifier

Derived property: Származtatott tulajdonság, az értékét más információkból kaphatjuk meg.

▪ Examples:



Derived property at an association end

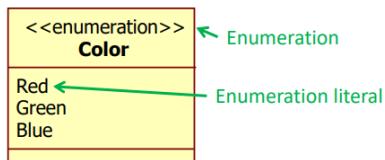


(Generalization between associations is also possible.)

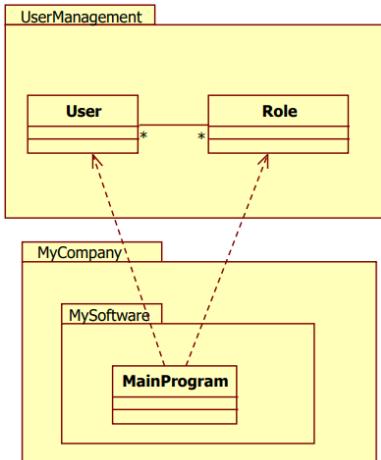
(Associations can also be derived. Notation: slash before the association name.)

◀ 40 ▶

Enumeration:



Packages:



Objektum orientáltság

Abstraction: A lényegtelen információk elhagyása. Valódi dolgokat objektumoknak feleltethetünk meg.

Classification: Egységes viselkedés és tulajdonságok alapján csoportosítani egy osztályba.

Encapsulation: Egy osztály ne adjon közvetlen hozzáférést az attribútumaihoz. Csak függvényen keresztül.

Inheritance: A származtatott osztály felhasználhatja az őse viselkedését. (Adatait nem!)

Polymorphism: A metódus hívójának nem kell azzal foglalkozni, hogy egy objektum egy őséhez tartozik e. Megoldás viruális metódusokkal és örökléssel.

Coupling: Annak a mértékegysége, hogy mennyire szorosan kapcsolódik két elem. Minél alacsonyabb annál jobb.

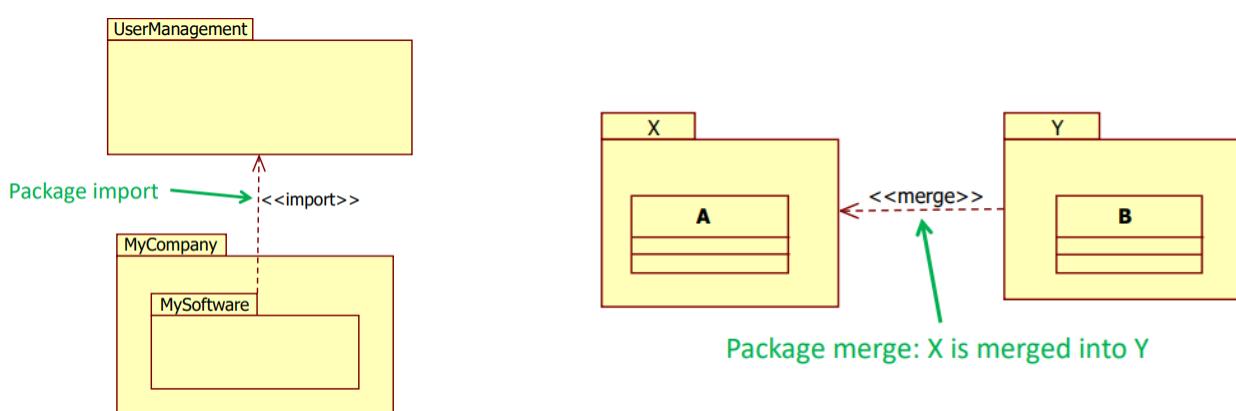
Cohesion: Annak a mértékegysége, hogy milyen szorosan kapcsolódnak egy elem részei. Minél nagyobb, annál jobb.

Package Diagram

Függőségeket mutat be package-ok között.

<<import>>: Az importáló hozzáadja az importált package elemeinek nevét a saját névteréhez. (import/include)

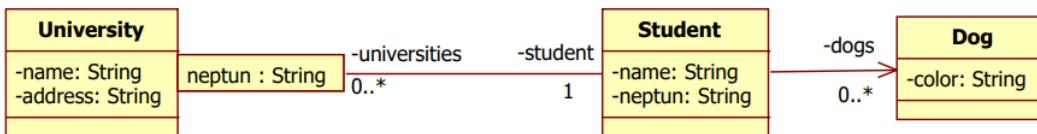
<<merge>>: Irányított kapcsolat két package között, ami azt jelöli, hogy a két package tartalmát össze kell vonni.



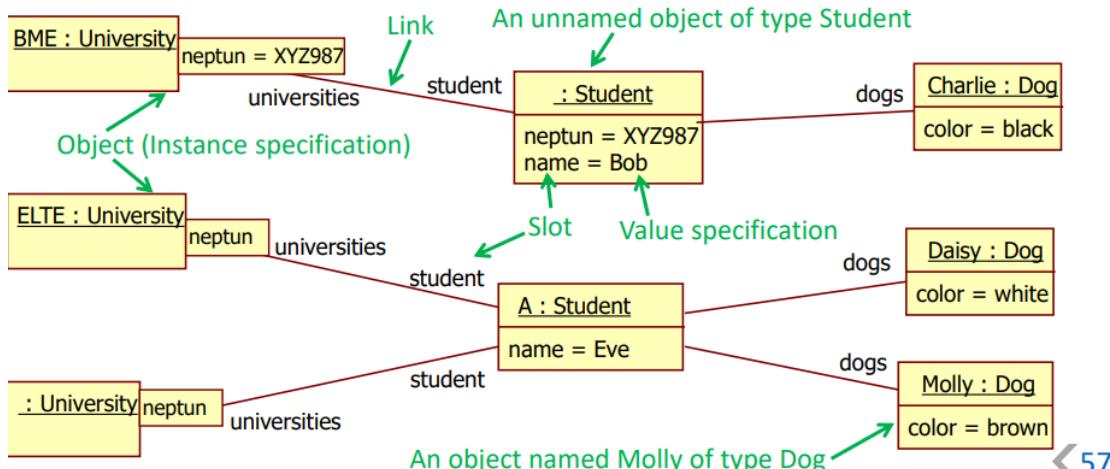
Object Diagram

Az osztály diagram egy példánya: osztály példánya (object vagy specifikáció), association példánya (link). A rendszer egy részletes állapotát mutatja be egy időpillanatban.

Class Diagram:



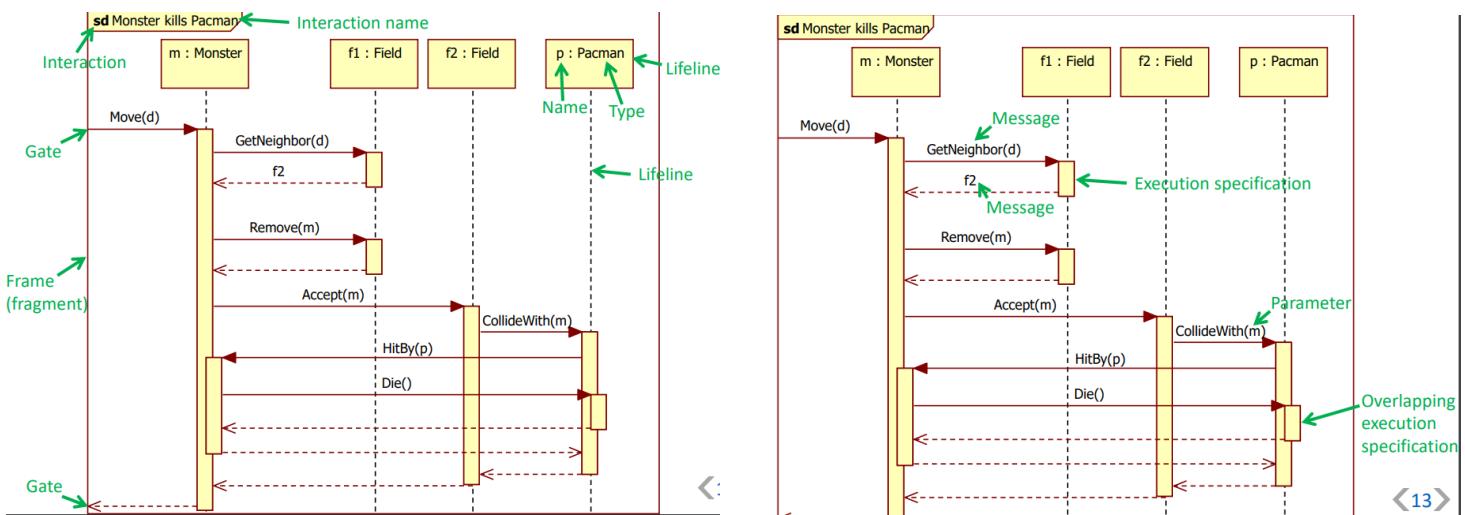
An Object Diagram for the Class Diagram:



57

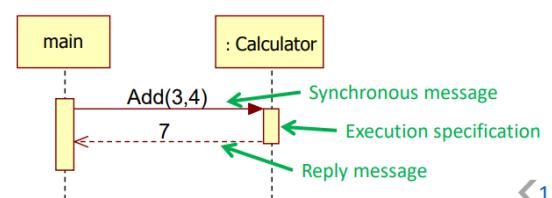
Sequence Diagram

Interakciók grafikus reprezentációja. A rendszer dinamikus viselkedését mutatja be. Az interakciók az információk továbbításának bemutatására koncentrálnak.



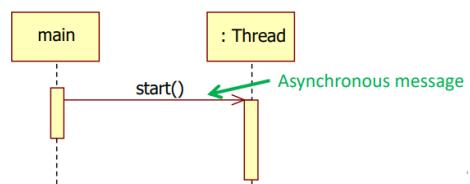
13

Szinkron üzenet: Szinkron metódus hívás. Egy végrehajtást indít el. A hívó vár amíg a hívott metódus befejeződik.



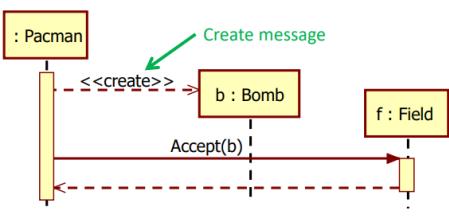
1

Aszinkron üzenet (signal): Aszinkron metódus hívás. Egy végrehajtást indít el. A hívó nem várja meg hogy a metódus befejeződjön.

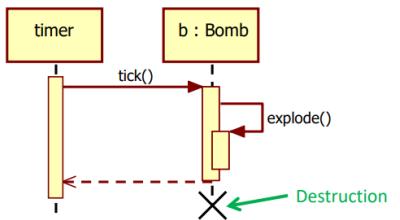


10

<<create>> üzenet: Létrehoz egy új lifeline-t.



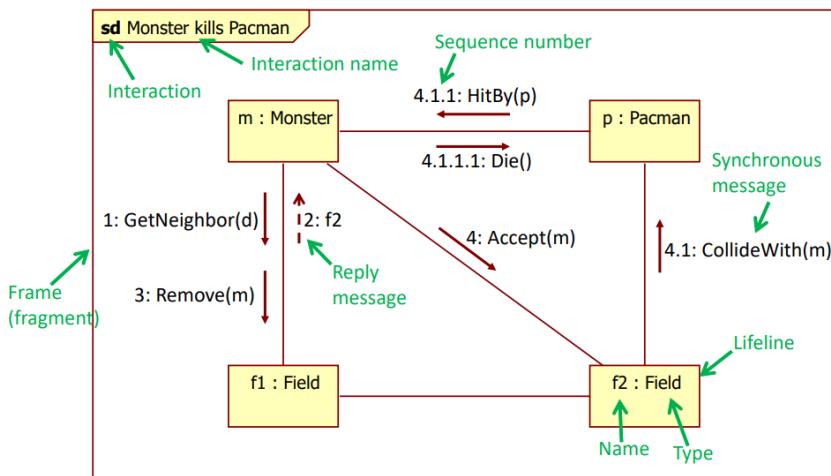
Destruction: Befejez egy lifeline-t.



Abbrev	Kind	Meaning	Abbrev	Kind	Meaning
alt	Alternatives	Represents a choice of behavior. At most one of the operands will be chosen.	neg	Negative	Represents traces that are defined to be invalid. All fragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible.
opt	Option	Represents a choice of behavior where either the (sole) operand happens or nothing happens.	critical	Critical region	The region is treated atomically by the enclosing fragment when determining the set of valid traces.
break	Break	Represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing fragment.	ignore	Ignore	Ignore designates that there are some message types that are not shown within this combined fragment.
par	Parallel	Represents a parallel merge between the behaviors of the operands.	consider	Consider	Consider designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be ignored.
seq	Weak sequencing	Represents a weak sequencing between the behaviors of the operands.	assert	Assertion	Represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.
strict	Strict sequencing	Represents a strict sequencing between the behaviors of the operands.	loop	Loop	Represents a loop. The loop operand will be repeated a number of times.

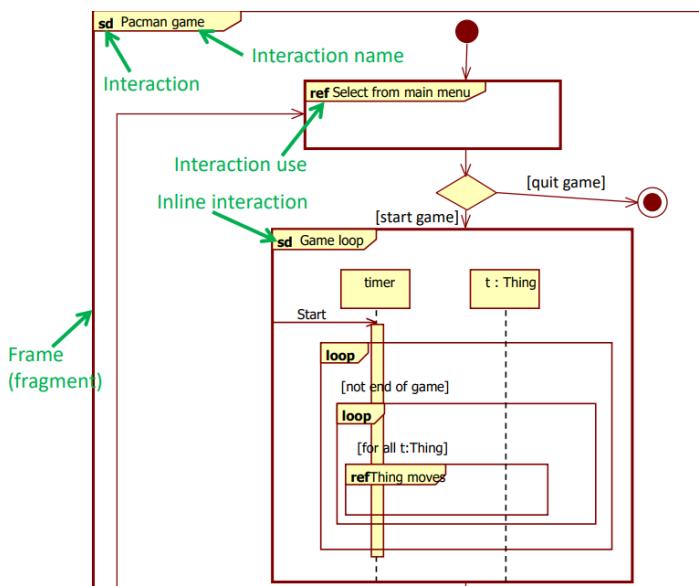
Communication Diagram

Interakciók grafikus reprezentációja. A rendszer dinamikus viselkedését mutatja be.



Interaction Overview Diagram

Activity diagramokon keresztül mutat be interakciókat, ami áttekintést ad az irányító folyamatról.



State Machine Diagram

Esemény vezérelt viselkedést mutat be.

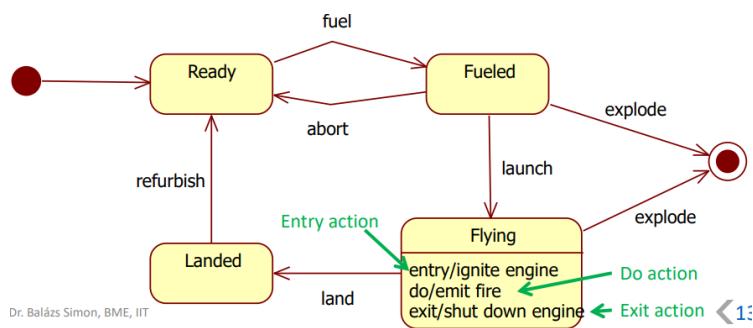
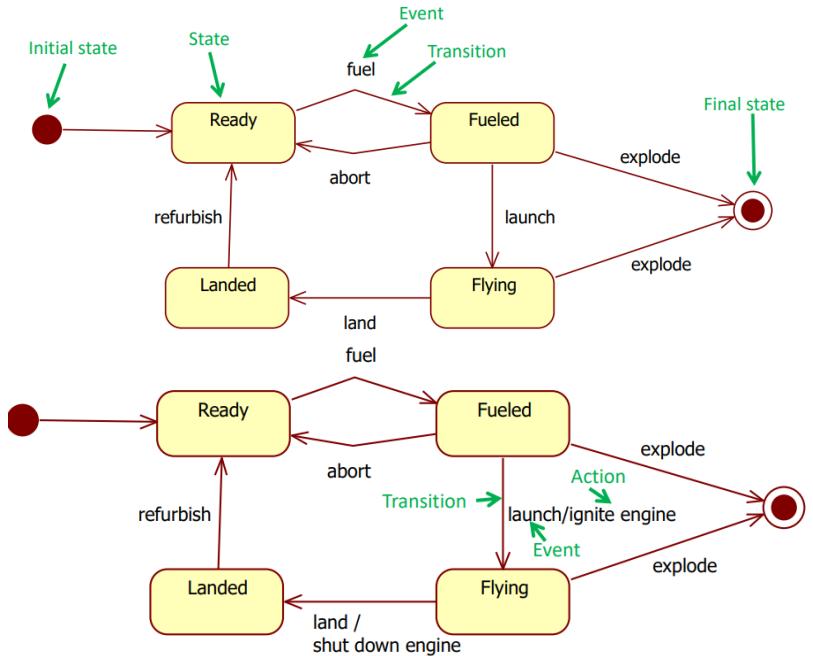
Egy objektum lehetséges állapotait modellezi, és hogy hogyan jut azokba az állapotokba.

State/állapot: A különböző információk kombinációja, amit egy objektum tartalmazhat.

Behavior state machine: A rendszer részeinek viselkedését mutatja be.

Protocol state machine: A rendszer részeinek interakciót és használt protokolljait elemzi.

A state machine eseményekre (event) valamelyen tevékenységgel (action) válaszol.

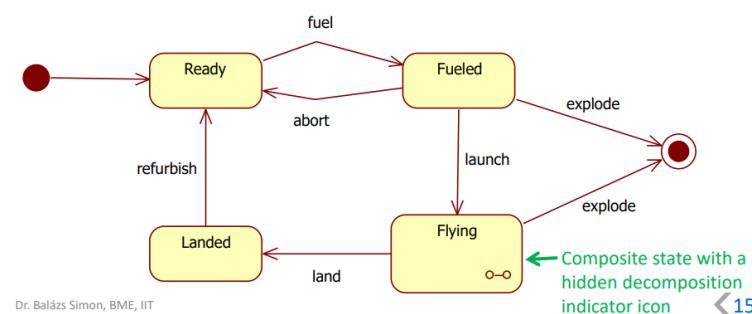
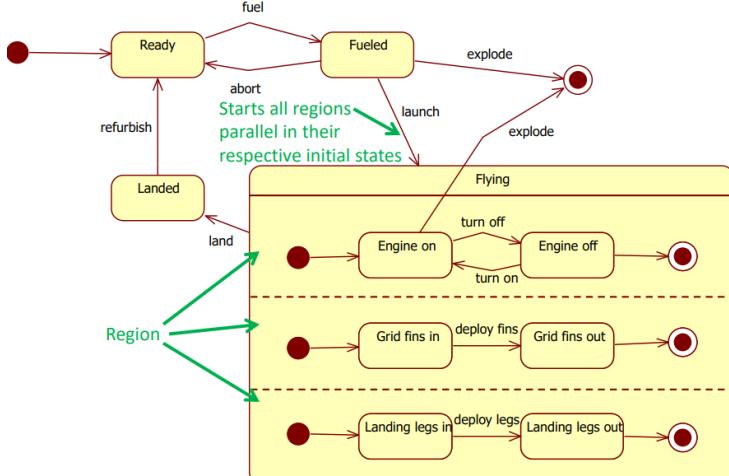


Entry action: Akkor megy végbe, ha egy külső transition belép az állapotba.

Exit action: Akkor megy végbe, ha elhagyják az állapotot.

Do action: Akkor kezdődik, ha belépnek az állapotba, az entry action után, és addig tart amíg nem lépnek ki az állapotból vagy véget nem ér a tevékenység.

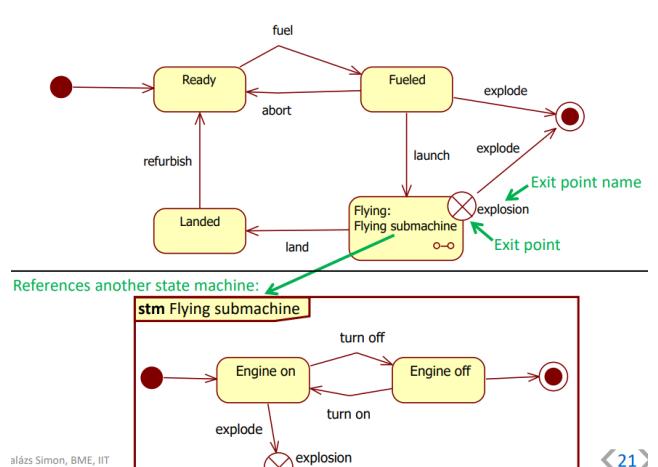
Composite state: Vannak régiói, a régióknak vannak állapotai, és függetlenek egymástól.



Default activation: Egy olyan átmenet, ami az összes régiót párhuzamosan elindítja.

Explicit activation: Ha egy átmenetet az egyik régió egyik állapotába megy, akkor csak azt indítja el, a többi default activated.

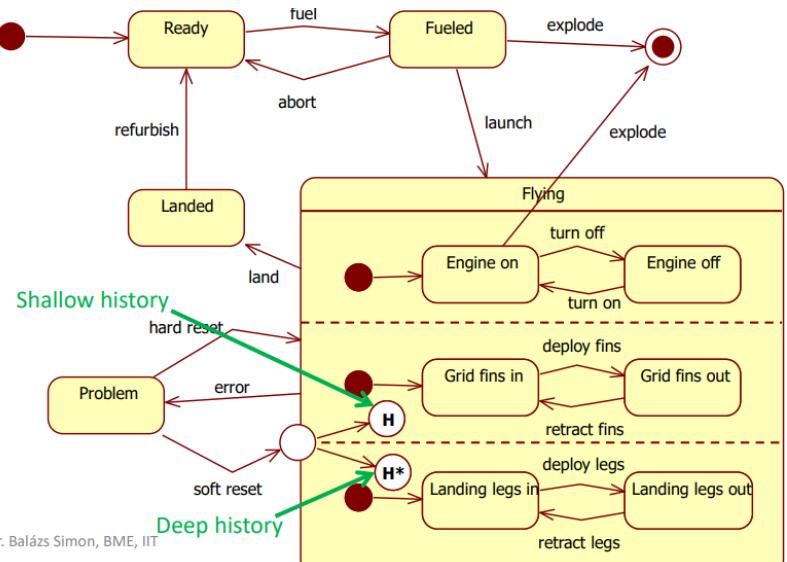
Submachine: Külön state machine, ami így többször felhasználható.



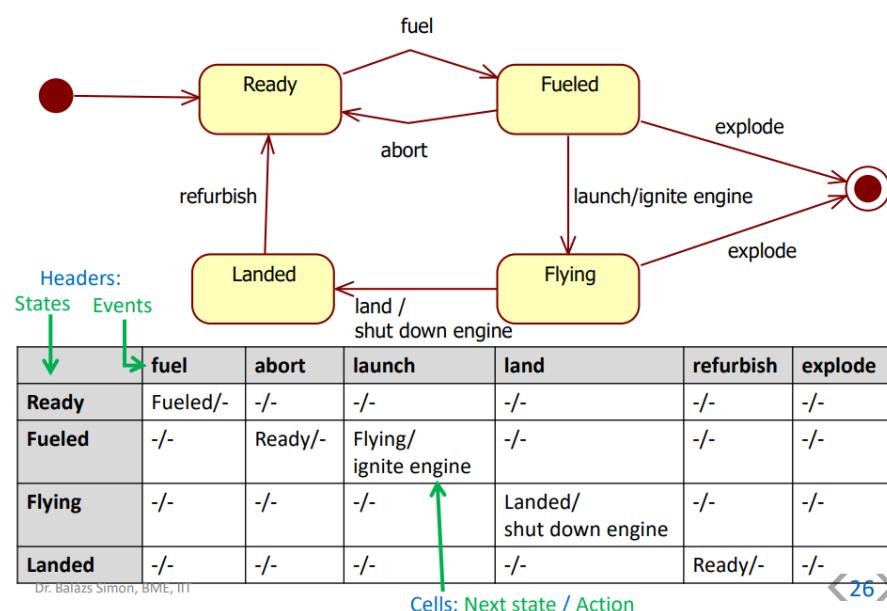
History state: segít visszaállítani azt az állapotot egy composite state régióiban, ami akkor volt mikor utoljára kiléptek belőle. Ha nincs előző állapot, mert először lépnek be a composite statebe, akkor, ha a historyból indul átmenet egy állapotba akkor initial state-ként viselkedik, ha nem indul átmenet, akkor default entryvel indul.

Deep history: minden állapotot visszaállít.

Shallow history: A legfelső composite state állapot konfigurációját visszaállítja, de az alállapotokat nem.

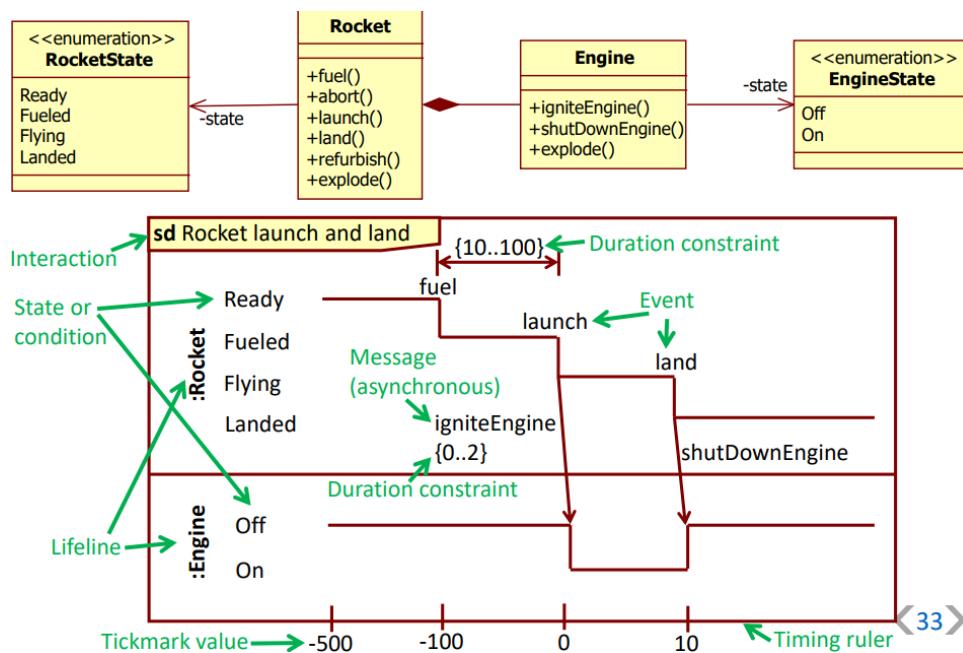


Tabular form:



Timing Diagram

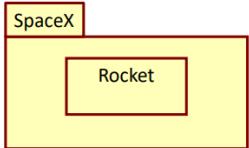
Az állapotok változására koncentrál lifeline-okon keresztül egy lineáris idősávban. Balról jobbra telik az idő.



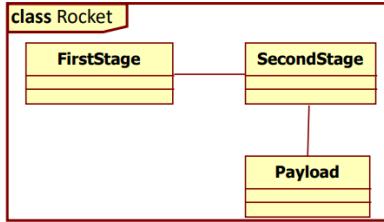
Composite Structure Diagram

Egy classifier belső struktúráját mutatja be.

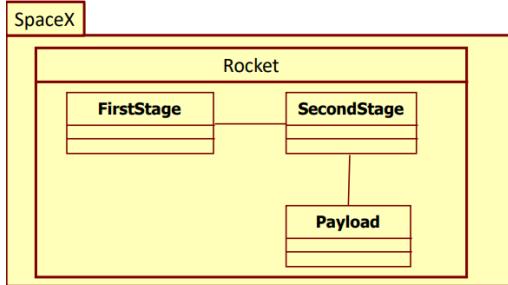
Class Diagram with no internal structure
for the Rocket class:



Composite Structure Diagram for the Rocket class:



Class Diagram with internal structure
compartment for the Rocket class:



«41»

Profile Diagram

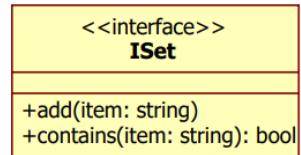
Egyedi stereotype-okat hozhatunk létre vele.

Stereotype: modell elemekhez lehet rendelni, megváltoztatja a jelentésüket.

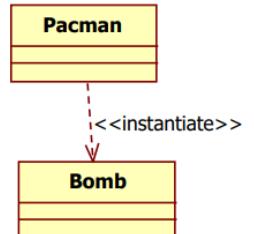
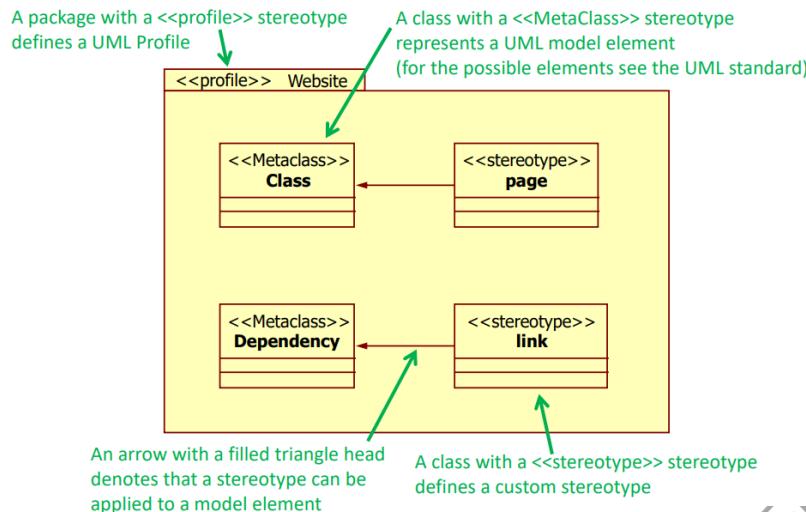
pl:

«interface»: osztályhoz rendelhető, azt mutatja, hogy az osztály interface

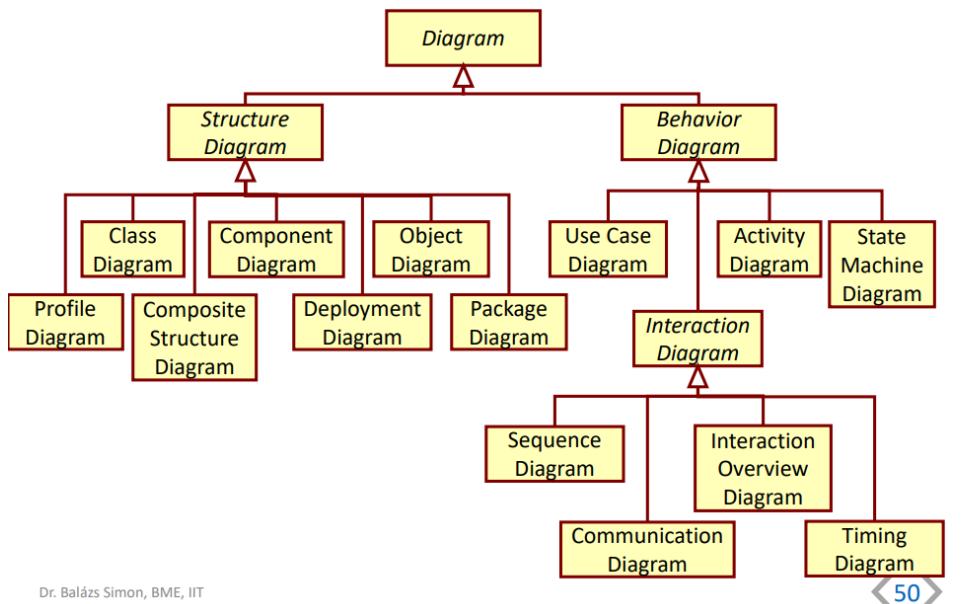
«instantiate»: függőséghez rendelhető, azt mutatja, hogy az egyik létrehoz egy példányt a másikból.



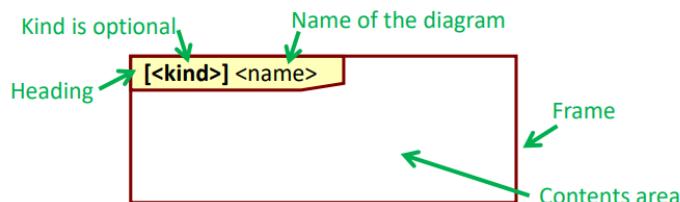
pl:



Összefoglalás



A diagrammoknak lehet keretük:



UML-en túl:

- Object Constraint Language (OCL): Szöveges szkript nyelv metódusok, feltételek és változók leírására.
- XML Metadata Interchange (XMI): XML formátum, diagramok eszközök közötti cserélgetésére.
- MetaObject Facility (MOF): UML standard leírására szolgáló modellező nyelv.

Objektumorientált tervezési elvek

A változás problémája:

A szoftver változhat. A jó tervezés ezt az átállást könnyebbé teszi.

Akkor van probléma, ha úgy módosulnak a feltételek, amire a tervezésnél nem számítottunk.

Későbbi változások szemben állhatnak az eredeti tervezési filozófiával, ezért fontos a dokumentáció.

A változás valószínűsége:

Nem minden egyértelmű, hogy van további ok változásra. Az is lehetséges, hogy a változás sose fog végbe menni.

A tapasztalataink és tudásunk alapján kell kiszámítanunk a változás valószínűségét.

Ne tervez változásra, ha alacsony a bekövetkezésének valószínűsége. (YAGNI= You Ain't Gonna Need It)

Egy tervezés rossz, ha:

- Nehéz a változtatás, mert a változás túl sok részét érinti a rendszernek. (Rigidity)
- A változás tönkreteszi a rendszer egy részét. (Fragility)
- Nehéz a rendszer bizonyos részeit újra felhasználni más alkalmazásokban, mert nem lehet kibogozni őket a jelenlegi alkalmazásból. (Immobility)

Rossz tervezés oka: Túl erős függőségek a rendszer részei közt.

Megoldás: Függőségek csökkentése és függőségek elhagyása a gyakran változó vagy problémásabb részeknél.

A SOLID alapelvek

A jó tervezés 5 alapelve:

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single Responsibility Principle (SRP)

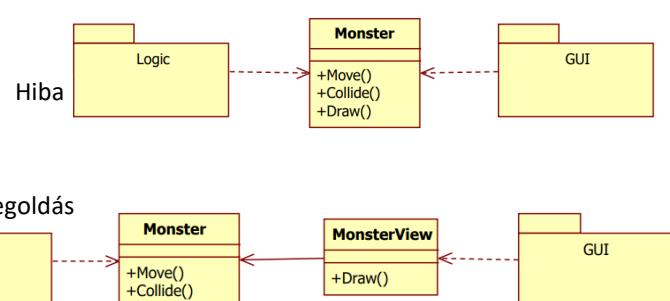
Egy osztálynak csak egy oka legyen a változásra. Itt: felelősség = a változásra való ok.

Ez azt jelenti, hogy ha egy osztálynak több mint egy felelőssége van, akkor több osztályra kell osztani.

Fel kell osztani a felelősségeket:

- Implementációs szinten: ha nem lehet szétválasztani őket
 - külön osztályok
 - egyik használja a másikat, de fordítva nem
- Interfész szinten: ha szét lehet választani őket
 - interfések a különböző felelősségeknek
 - minden interfész implementálása az osztályban

Előny: A függőségek a problémás részektől elfele tartanak.



Open/Closed Principle(OCP)

A szoftver elemek (osztályok, modulok, funkciók...) nyitottak legyenek bővülésre, de a módosítást ne támogassák.

Nyitottak legyen bővülésre: a modul viselkedését ki lehessen bővíteni úgy, hogy kielégítse a megváltozott feltételek.

Módosítást ne támogassák (zártak legyenek módosításra): A viselkedésének bővítése ne változtassa meg a modult.

OCP lényege: Készülj fel a változásra.

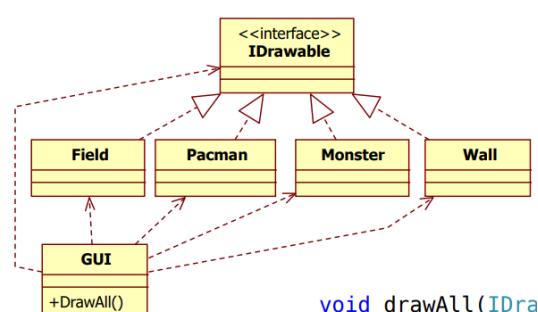
Bővíthető legyen: Új alosztályok, overriding metódusok, polimorfizmus, delegáció...

Az eredeti kód ne módosuljon. Csak bug-ok javítása.

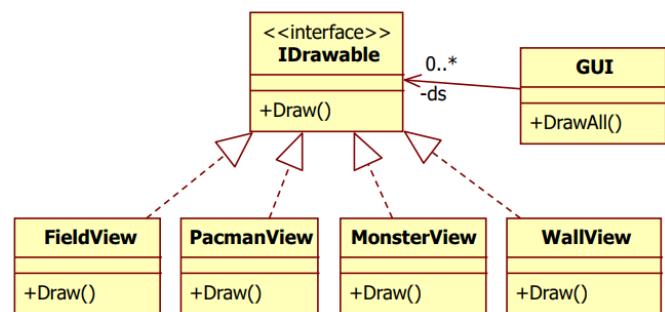
Összefoglalva: A kódot új kód hozzáadásával bővítsd, ne az eredeti megváltoztatásával.

A sorrend meghatározása legyen elkülönítve az osztályok felelősségeitől.

Hiba:



Megoldás:



Liskov Substitution Principle (LSP)

Altípusoknak megfeleltethetőknek kell lenni az alaptípusaikkal.

Minden származtatott osztálynak behelyettesíthetőnek kell lennie ott, ahol az alap (Base) osztály használva van, anélkül, hogy a felhasználónak tudni kéne a különbséget.

LSP megszegése: Explicit típus ellenőrzés (is..., instanceof, dynamic_cast, stb)

Pi: square-rectangle problem (matematikailag a négyzet egy téglalap, de máshogy viselkedik, és ez számít az OO-nál)

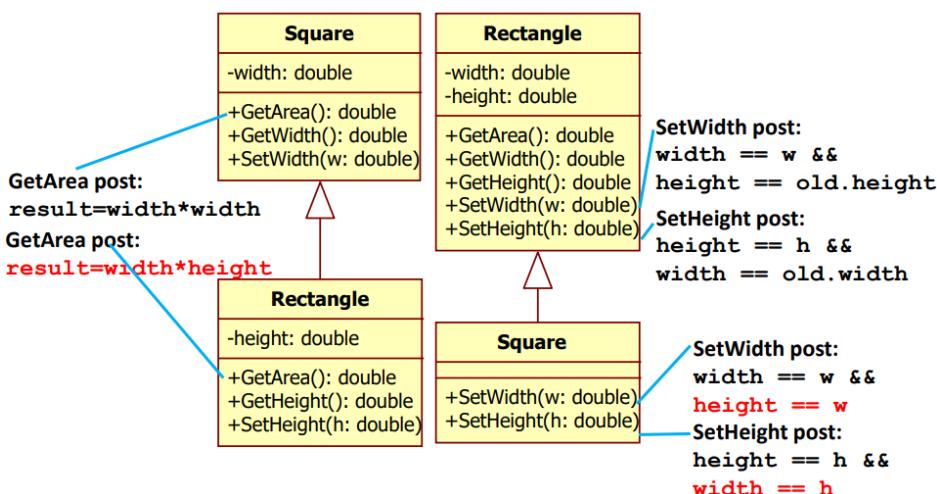
```
void Draw(Shape s) {
    if (s instanceof Rectangle) DrawRectangle((Rectangle)s);
    else if (s instanceof Ellipse) DrawEllipse((Ellipse)s);
}
```

LSP megszegése az OCP megszegéséhez vezet.

Design-by-Contract:

- A származtatott osztály hirdetett viselkedése megfeleltethető a Base osztályéval.
- Contract: a hirdetett viselkedés
 - Előfeltételek/pre-conditions: Be kell teljesülniük metódushívás előtt. A hívónak biztosítani kell, hogy fennmaradjanak.
 - Utófeltételek/post-conditions: Be kell teljesülniük metódushívás után. Az implementáló osztálynak biztosítani kell, hogy fennmaradjanak.

Szabály: A származtatott metódus lecserélheti az eredeti előfeltételt egy ugyanolyanra vagy gyengébbre, és lecserélheti az utófeltételt egy ugyanolyanra vagy erősebbre. (Expect no more, provide no less.)



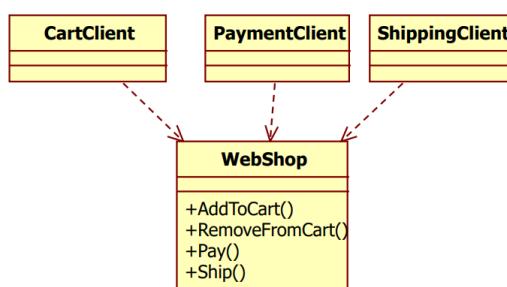
Interface Segregation Principle (ISP)

A klienseknek nem kéne olyan metódusoktól függeni, amiket nem használnak, csak olyanoktól, amiket igen.

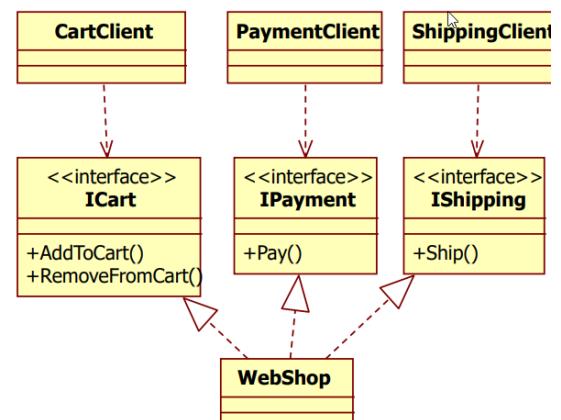
A nagy interfészket fel kell bontani több kliens-specifikus interfészre.

Az osztály implementálja az interfészket. A kliens csak az interfészektől függ, amit használ. Így a kliensek függetlenek lesznek egymástól.

Hiba:



Megoldás:



Dependency Inversion Principle (DIP)

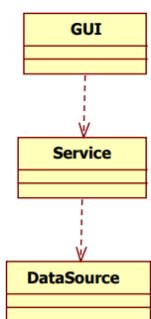
Magas szintű modulok ne függjenek alacsonyabb szintű moduluktól.

Absztrakció ne függön a részletektől, a részletek ne függjenek absztrakcióktól.

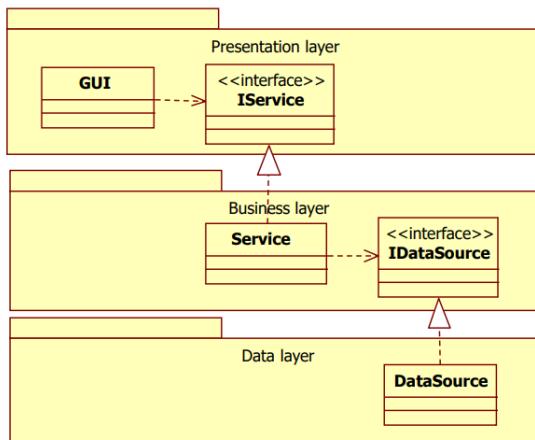
Az alkalmazás modulokból áll. Általában előbb írják az alacsonyabb szintű modulokat majd a magasabb szintűekkel összevonják őket. De ez azt jelenti, hogy az alacsonyabb szintű modulok megváltozása hatással van a magasabb szintűekre.

DIP lényege: Az alacsonyabb szintű modulok függjenek a magasabb szintű moduluktól.

Hiba:



Megoldás:



További elvek:

Don't Repeat Yourself (DRY)

Minden eltárolt információnak egy darab félreérhetetlen, hiteles reprezentációja kell legyen a rendszeren belül.

Ismétlődések csökkentésére koncentráljunk.

Azért rossz a duplikáció, mert, ha valami megváltozik egy repetitív kódban, azt az összes előfordulásnál javítani kell.

Single Choice Principle (SCP)

Ha egy rendszernek egy csoport alternatívát kell támogatnia, az az előnyös, ha csak egy osztály ismeri az összes alternatívát.

Tell, don't ask (TDA)

Metódusokat úgy kell meghívni, hogy előtte a cél objektum állapotát nem ellenőrizzük le.

Ha szükséges, akkor a cél objektum feladata az állapotellenőrzés.

TDA megszegése azt jelenti, hogy a felelősségek nem a megfelelő helyeken vannak, és a DRY megszegését is jelenti.

Law of Demeter (LoD)

Egy objektum metódusa csak a saját, a saját paraméterei, a tagjai, vagy az általa létrehozott objektum függvényeit hívhatja.

Egy metódus nem hívhatja más objektumok tagjainak metódusait.

Megoldás: minden objektumban legyen olyan metódus, ami a hívást átadja a következő objektumnak (csak saját tagoknak).

Design Pattern

Egy általános megismételhető megoldás a gyakran előforduló problémákra.

Bevezetés

A szoftverről és a szoftverfejlesztésről. Aktuális trendek és problémák. Folyamatok és modellezésük.

Definíciók

- **IT - Information Technology:** Bármire vonatkozhat, ami számítógépes technológiával kapcsolatos, akár még emberre is. Példák: hardver, szoftver
(IT job pl: programozó, szoftvermérnök, tesztelő, rendszergazda...stb)
- **ICT - Information and Communication Technology:** Az Információ és Kommunikáció Technológia, az infokommunikáció értelmezését jelenti, de az adatfeldolgozás mellett az adattovábbítással kapcsolatos műveleteket is magába foglalja.
Pl: felhő alapú adattárolás, IT infrastruktúra kialakítása, Szerver szolgáltatások... stb
- **Szoftver:** Számítógépes programok, folyamatok és esetlegesen a számítógépes rendszer üzemelésére vonatkozó dokumentációk és adatok. A szoftver megjelenhet koncepciók, ügyletek vagy eljárások alakjában.
Pl: számítógépprogram
- **Szoftvertermék:** Számítógépi programok, eljárások, adatok és a hozzájuk tartozó dokumentáció komplett készlete. A szoftvertermék tartalmazza a felhasználói adatokat.

Másfajta megfogalmazások:

- Szoftver = olyan szellemi alkotás, amely magában foglalja valamely adatfeldolgozó rendszer működését biztosító programok, eljárások, szabályok és a hozzájuk tartozó dokumentáció összességét. VAGY Parancsok sorozata, amely a hardver működését ellenőrzi.
- Szoftvertermék = számítógépi programok, eljárások, adatok és a hozzájuk tartozó dokumentáció olyan komplett készlete, amelyet valamely felhasználónak való szállításra terveztek.

Problémák

Hibás szoftver:

- Emberi életet veszélyeztethet
- pénzügyi csödöt eredményezhet
- világkatasztrófához vezető folyamatokat indíthat le

Mégis a szoftverben hibák vannak és projektek nagy része problémákkal küzd.

Szoftvernek jónak kell lennie. -> Normális elvárás, hogy a szoftvertermék a legmagasabb szinten: elégítse ki a felhasználó követelményeit és rendeltetésének megfelelően működjék.

Korábban: 1 fejlesztő - 1 felhasználónak -> Cél: program fusson le, várthoz hasonló eredményekkel.

Később: több alkotó – több felhasználónak -> Cél: érthetőség, hordozhatóság, tanulhatóság.

Mai cél: Hatékonyság, megbízhatóság, hibamentesség, adatok biztonsága, adatvesztés nélkül.

Mitől hibás?

Szoftvert emberek írják. Tapasztalt programozók átlagban minden 7-10 sorban vétenek 1 hibát.

Ezeknek a felét a gépyelvre történő fordításkor kijavítják.

A tesztelés során további hibák kijavulnak, de hibák **15%-a** meg marad.

Hibák okai:

1. Az emberi tényező
2. Kommunikációs hiba: A követelmények összegyűjtésére, egyeztetésére, dokumentálására nem szánnak elég időt. Egy fejlesztő megpróbál egy másik fejlesztő által fejlesztett kódot módosítani több-kevesebb sikkerrel.
3. Irrealis fejlesztési idő: Nagyon gyakran a szoftvert irreálisan rövid idő alatt vállalják kifejleszteni, kevés erőforrással. Ilyenkor a követelmények betartását nem veszik eléggé szigorúan, mert fontosabb, hogy határidőre átadjanak valamit.
4. Rossz tervezés
5. Rossz kódolási gyakorlat
6. Verziókezelés hiánya
7. Hibás, harmadik féltől származó eszközök
8. Képzett tesztelők hiánya
9. Változtatások az utolsó percben

Nemrégiben előfordult hiba: BKK és T-Systems Hungary esete

Feladatmegoldás folyamata

1. *Olvassuk el a feladatkiírást*
2. Értsük meg a feladatot
3. Tervezzük meg a megoldást
Használunk fel korábbi tapasztalatokat, meglévő megoldásokat, rendelkezésre álló elemeket.
4. *Implementáljuk a tervet (kódolunk)*
5. Teszteljük a megoldást
Javítsunk hibákat, amíg nem találunk más hibát (újratesztelés)
6. *Adjuk át a megoldást*

(Amire nem szabad időt vesztegetni: döltbetűs lépések)

Minél nagyobb a feladat, annál több elemet kell észben tartani.

Módszerek, számárvezetők, útmutatók, minták **szükségesek**.

Nem engedheti meg magának senki, hogy rossz szoftvert fejlesszen.

Lehet nulla hibával rendelkező szoftvert fejleszteni, csak drága. (pár ezer dollár soronként)

Mitől „más” a szoftver?

Nincs fizikai léte.

Előállításának saját életciklusa van.

Gyorsan változik és alkalmazkodnia kell az új hardver-szoftver környezetekhez.

Felhasználók nem tudják megfogalmazni mit akarnak, de mégis magas elvárásai vannak.

Nehéz meghatározni és mérni a szoftver minőségi jellemzőit, mert:

- A szoftvernek alkalmazkodnia kell a minden változó hardver követelményekhez.
- A minőség még egyazon termék esetében sem állandó.
- A szoftvernek nincs a hagyományos módon mérhető, fizikai léte.
- A szoftver minősége függ a minőséget értékelő személyétől

A szoftverfejlesztés

Komplex feladat, emiatt projektben végezzük, ami átnyúlhat országokon, kontinenseken is.

Sokféle szakértelem kell:

- Üzleti terület
- Kommunikáció, Csapaton belül is
- Projekt tervezés, követés, vezérlés
- Szoftverfejlesztési módszerek és eszközök

Legyünk nyitottak, dolgozunk együtt másokkal, tanulunk új dolgokat másoktól.

Használjuk fel módszertanokat, modellek, szabványokat, de úgy, hogy nekünk hasznosak legyenek.

Osszuk meg tapasztalatainkat, ezáltal tanulunk mások hibáiból.

Fegyelmezetten fejlesszünk!

Szoftvermérnökség több definíciója:

- Több személy által, több verziójú szoftver fejlesztése
- Egy mérnöki diszciplína, mely magas minőségű szoftver-rendszerek költség-hatékony fejlesztésére törekszik.
- Számítógépes eszközök alkalmazása problémák megoldására.
- Egyfajta mérnökség, amely a számítástechnikai tudomány elveit és matematikát alkalmaz szoftveres problémák költség-hatékony megoldására
- Módszeres, fegyelmezett, számszerűsíthető megközelítés, mely szoftver fejlesztésére és karbantartására irányul; azaz: a mérnöki tudományok szoftverre történő alkalmazása

Kapcsolódik a következőkhöz a szoftverfejlesztés:

- Folyamatok
 - Fejlesztési/műszaki, menedzsment, támogató, szervezeti...
 - Módszerek/módszertanok: folyamatok csoportja, melyek bizonyos szempontok szerint összetartoznak.
- Termékek (Jellemzők, mérőszámok)
- Emberek (képességek, csapatmunka)
- Projektek

Szabályok, amiket érdemes betartani:

Tiszteljük a többieket és legyünk korrektek és becsületesek!

Legjobb képességünk szerint végezzük el a feladatot!

Tartsuk tiszteletben a törvényt!

A „Szoftvermérnöki szabály”: **Nincs abszolút út!** – Helyette sok megközelítés

Ezekből válogatva több segítséggel tudjuk összerakni a saját megoldásunkat.

Vannak általánosan érvényes „tudás-elemek”, amelyeket meg kell jegyezni és alkalmazni kell.

Nem kell minden feltalálni a spanyolviaszt -> DE meg kell ismernünk a meglévő megközelítéseket és meg kell tanulnunk megfelelően alkalmazni őket.

A szoftvermérnökség sokkal több a kódolásnál! -> szoftver minősége több a kód minőségénél!

Határozzuk meg folyamatainkat és alkalmazzuk is, akkor is ha éppen pánikba estünk, mert szorít az idő!

Modellezük a szoftverfejlesztést, mert az egy lehetséges kockázatcsökkentési tevékenység és működik!

Kezdetek és a múlt:

Szoftverfejlesztés kb 50 éves. A szoftver életünk fontos eleme lett.

Nagyon gyors fejlődésben ment át és befolyásolta a különböző tudományágakat és művészeteik egész életét.



A szoftverfejlesztés modellezése

A szoftver jó minősége mindenkor is fontos volt, de nem lehet általánosan meghatározni.

Minden modelleztek az elmúlt 50 év során: Terméket, Folyamatokat, Erőforrásokat, definíciókat és metrikát határoztak meg. (pl röntgen, szobrok, rajzok)

Folyamatok:

- **Projektmenedzsment folyamatok**
 - Projekttervezés, -követés és -vezérlés, kockázatkezelés
- **Támogató folyamatok**
 - Konfigurációmenedzsment, minőségbiztosítás, mérés
- **Szervezeti folyamatok**
 - Stratégiai tervezés, szervezeti képzés, folytonos folyamatfejlesztés
- **Fejlesztési folyamatok**
 - Követelményfejlesztés, tervezés, kódolás, tesztelés, karbantartás
 - Életciklus modellek

Szoftvercégek által alkalmazott folyamatokat csoportokba sorolják -> folyamatfejlesztési modellek támogatják (SPI)

Folyamat:

- Egy folyamat egy lépéssorozat és döntések sorozata, melyek a munka elvégzéséhez vezetnek, és megadják az elvégzés módját
- Akciók és műveletek sorozata, melyek eredményre vezetnek. Különösen a gyártásban használt fogalom.
- Egymással kölcsönhatásban álló tevékenységek sorozata, mely bemeneteket kimenetekké alakít, egy adott cél elérésére.

Folyamatmodell:

„Alapvetően minden modell hibás; némelyikük hasznos.” - George E.P. Box

3 fontos tulajdonság:

- Reprezentáció: a modellek a valóságot tükrözik
- Redukció: a modellek redukálják a valóságot, csak bizonyos aspektusaival foglalkoznak
- Pragmatikus tulajdonság: a modelleket bizonyos céllal hozzák létre

Definíció:

- Egy olyan keretrendszer, amelyben hasonló folyamatok egy csoportját írják le, pl. egy teszt fejlesztési modell.
- Annak leírása, hogy milyen tevékenységeket és milyen sorrendben kell végrehajtani ahhoz, hogy a kitűzött célt elérjük.

Leírása:

Többféle formában történhet: Szövegesen, diagramokkal, sajátos jelölések.

Sokféle folyamatmodellezési nyelv, eszköz, szabvány... van.

Különböző részletezettségűek – rendszer, komponens, funkció szintű.

Más elemekre koncentrálnak, hogy segítsék a konkrét elemekre való modellezést. -> emberek, szerepkörök, feladatak...

Modellezések lehetőségei

UML aktivitás diagram

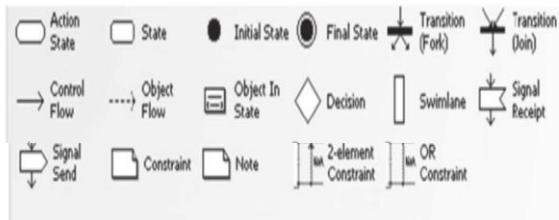


Figure 17 – Elements of UML activity diagrams, source: MS Visio

EPC diagramok

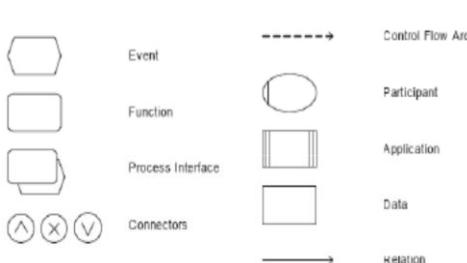
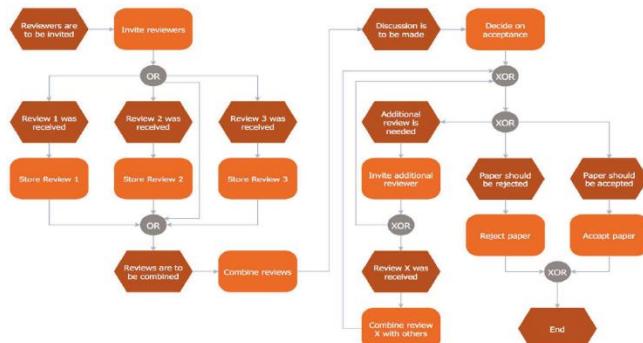


Figure 16 – Symbols of EPC diagram, source: (Mendling & Nüttgens, 2005)



BPMN, BPML

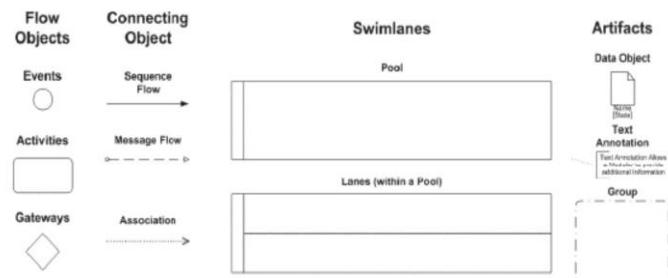
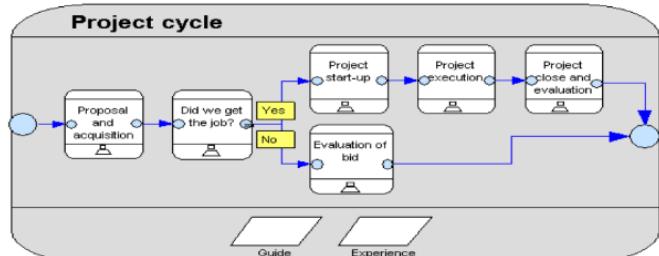


Figure 15 – Basic symbols of BPMN, source: (OMG, 2012)



Folyamatok elemei:

Egyéni folyamatokra koncentrál.

Input és output.

Folyamat célja, tevékenységei, felelősségei és hatáskörei.

Ellenőrzési pontok.

Teljesítmény- elvárások, szükséges kompetenciák.

Folyamat jellemzők.

További folyamat-jellemzők (pár példa):

Egyéni folyamatokhoz tartozó eljárásokat végrehajtják-e?

Folyamatot megfelelően terveztek-e?

Követik-e a tervet? - Hoznak-e Korrekciós intézkedéseket?

Ellenőrzik-e a végrehajtást?

Szabványosított-e a folyamat?

Értik-e a folyamatot mennyiségileg is?

Folyamatosan javítják-e a folyamatot?

Egy érett szoftverfolyamat: Meghatározott, dokumentált, tervezett, követett, mért, ellenőrzött, hatékony, alkalmazásban van, alkalmazását oktatják, szabványos és javulásra képes. Ezek közül minél több elem van helyén, annál magasabb képességi szintű az adott folyamat.

Életciklus modell

Software Engineering Processes – szoftverfejlesztés műszaki folyamatai

Szoftverfejlesztés az alábbi folyamatokat tart:

- Követelmények fejlesztése és menedzsmentje
- Tervezés
- Kódolás
- Tesztelés
- Átadás
- Karbantartás

Tevékenységeket szoftverfejlesztési életciklus modellek írják le. (SDLC)

3 alapvető igazság:

- Kezdetben nem tudni előre minden.
- Garantált a változás.
- Mindig több feladat lesz, mint pénz és idő.

Szoftver életciklus

Egy időperiódus, ami akkor kezdődik, amikor a szoftverterméket kigondoljuk és akkor fejeződik be, amikor a szoftvert már nem használják. A szoftver életciklus jellemzően a következő fázisokat tartalmazza: koncepció, követelmény, tervezés, megvalósítás, teszt, installáció és ellenőrzés, operáció és üzemeltetés, valamint időnként egy leállítási fázist.

Kategóriák:

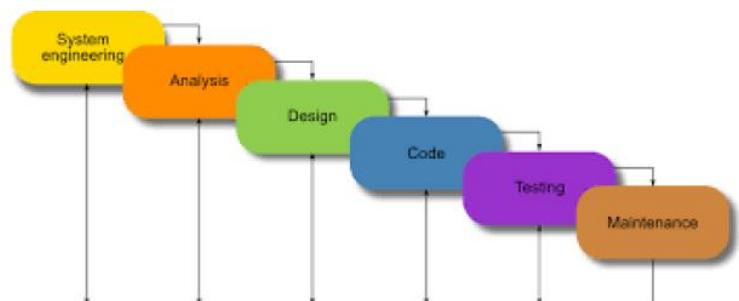
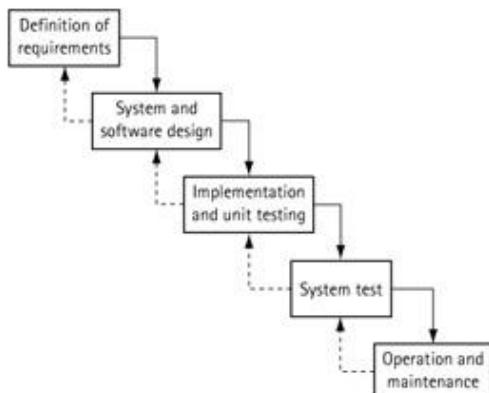
- Szekvenciális modellek
- Inkrementális modellek
- Iteratív modellek

A szoftverfejlesztési modelleket konkrét projekt és termék jellegzetességeinek függvényében testre kell szabni. Módszertanok mutatják hogyan lehet a problémákat megoldani és irányt mutatnak korábbi problémák tapasztalatival. Nagyobb valószínűséggel leszünk sikeresek, ha követjük ajánlásait.

Szekvenciális életciklus modellek

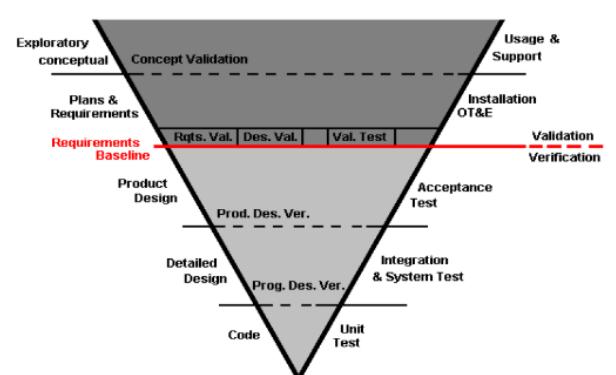
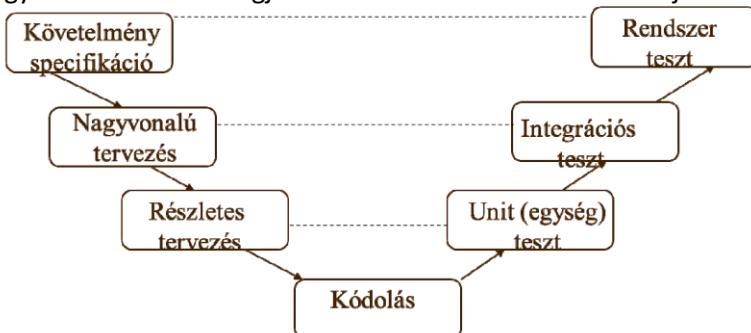
A szoftverfejlesztés folyamatát lineáris, szekvenciális folyamatként ábrázolják -> minden fázis csak akkor kezdődhet, ha előző befejeződött.

Vízesés modell



V modell

Egy adott fázisban megjelenített információkat felhasználjuk az adott fázishoz tartozó teszt esetek létrehozásában.



A V-modellben a rendszer validálása a fejlesztési életciklus végére kerül.

Explicit módon megjelenő tesztek a leírásában: Unit teszt, Integrációs teszt, Rendszerteszt, Verifikáció és validáció

Inkrementális modellek

A követelményeket kisebb csoportokra bontják és erre tesztelnek, kódolnak, terveznek. A kisebb csoportokat inkrementumoknak nevezzük.

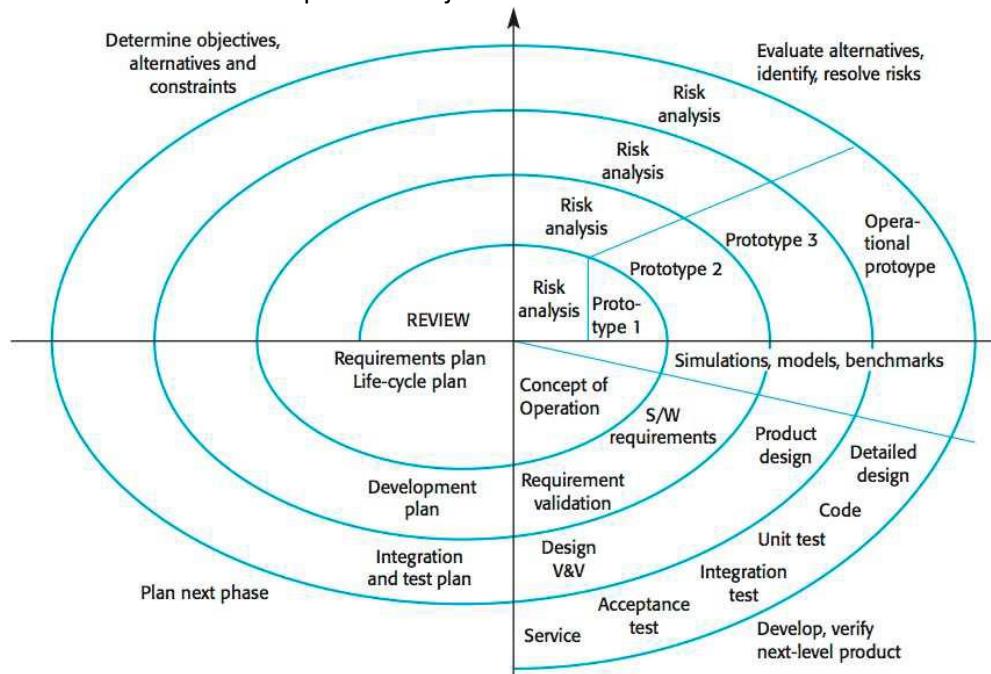
Regressziós teszt: ami az előző build-ben működött, a jelenlegiben is működik

Progressziós teszt: feltételezzük, hogy az integrációs lefutott és az új funkciókat teszteljük

pl:

- RUP: iteratív, inkrementális, use-case vezérelt (OO)
- RAD
- Spiral

- Boehm spirál modellje:



Iteratív modellek

Ciklikusak: előbb a követelmények kisebb csoportját implementáljuk rövid idő alatt, ezt iterációnak nevezzük.

Tipikusan **Agilis megközelítések** (Scrum, Kanban, XP, Lean)

Evolúciós fejlesztés: Első build-et definiálják, majd a felhasználó igényei alapján egészítik ki újabb elemekkel. Az evolúciós fejlesztés (rapid prototyping) egyik előnye, hogy a felhasználói visszajelzések viszonylag korán megjelennek a fejlesztési folyamatban.

RUP: iteratív, inkrementális, use-case vezérelt (támogatja az OO fejlesztést)

TDD: Tesztvezérelt fejlesztés (Test Driven Development) – Később részletesebben (7. előadás)

Extreme Programming

Gyors, változékony, rugalmas projektek módszertana -> könnyed és rugalmas

Ezekre épít:

- Egyszerűség
- Kommunikáció
- Visszajelzés
- Bátorság (új dolgokkal szemben)

User story-kra (később lesz róla szó részletesebben) épít és Spike-oka(megoldás) használ ismétlődő feladatokra.

Tervek elkészülése után iteratív módon fejlesztik.

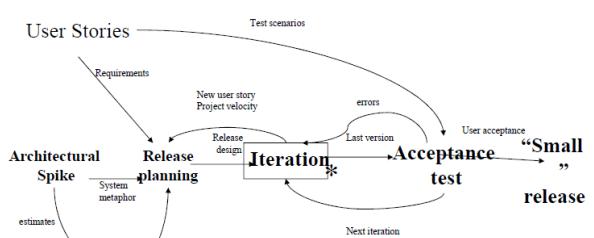
Együtt van az ügyfél, menedzser és a programozó.

Életciklus modell és programozási módszer is!

Scrum

Sikeres agilis eszköz (11. előadás)

Menedzsment eszköz, melynek vannak szoftverfejlesztéshez kapcsolódó elemei



Szoftver életciklus modellek

Mindegyik életciklus modell ugyanazokból a folyamatokból épül fel -> ezekből válogathatunk ISO-12207 szabvány E szabványnak való megfelelés azt jelenti, hogy folyamatokat, tevékenységeket és feladatot a szabványból választottak ki.

Hagyományos (vízesés) VS agilis megközelítés

Agilis	Vízesés
Folyamatos visszajelzés	Feedback at the end of the development
Minimális dokumentáció	Részletes dokumentáció
Reagálás a változásokra	A terv követése
Minden iteráció minden munkafázist tartalmaz	A fázisokat szigorúan csak egymás után lehet végrehajtani
Nem ismert minden specifikáció	Minden funkciót előre specifikálni kell
A fázisok nem különállóak. Az egész csapat ismer minden feladatot. A folyamatok párhuzamosan futhatnak.	A különböző fázisokban különböző szakemberek vesznek részt, akik különböző típusú tudással rendelkeznek

Hagyományos:

Általában egy sor, egymás utáni lépésekből áll

Formálisabb és nehézkesebb -> több dokumentációt igényel

Felhasználónak részletes elképzelése kell, hogy legyen.

Agilis:

Iteratív fejlesztési modellen alapul.

Kevésbé formális -> a felhasználó is részt vesz a fejlesztésben (követelmények nincsenek teljesen tisztázva).

Másfajta munkavégzés, ezért nem való mindenkinél.

Kreatívabb, nagyobb mozgástér, élénkebb kommunikáció jár hozzá, de emellett több felelősséggel, önfegyelem és valódi csapatmunka szükséges a munka megfelelő elvégzéséhez.

Agilis szoftverfejlesztés

12 alapelv az agilitáshoz: (ezeket betartva agilis)

1. A legfontosabb, hogy a vevő elégedett legyen.
2. Elfogadjuk, hogy a követelmények változhatnak.
3. Gyakran szállítsunk működő szoftvert. (akár hetente vagy havonta)
4. Megrendelővel együtt dolgozzanak a fejlesztők minden nap.
5. Építünk motivált egyénekre. Teremtsük meg nekik a megfelelő környezetet.
6. Személyes beszélgetés a leghatásosabb és leghatékonyabb a fejlesztő csapaton belül.
7. Előrehaladás mércéje a működő szoftver
8. Az agilis módszertanok elősegítik a fenntartható fejlesztést. Egyenletes sebességet meg kell őrizni.
9. Folyamatos figyelem a technikai kiválóságra és a jó tervezésre fokozza az agilitást.
10. Egyszerűség – az el nem végzett munkamennyiséget maximalizálásnak művészete.
11. Fontos az önszerveződő csapatmunka
12. A fejlesztő csapat rendszeres időközönként megfontolja a hatékonyabbá váláshoz szükséges dolgokat.

Jól kidolgozott gondolatrendszer, ami az inkrementális fejlesztést veszi alapul.

Gondolkodásmód:

Fogadjuk el a változtatásokat! A szoftver folyamatosan fejlődik; a felhasználó állandó visszajelzést ad. Érték-vezérel.

Folyamatos az átadás és minden átadásnál egy kis résszel egészül ki a szoftver.

Mindig kicsivel javul a szoftver. Agilisan dolgozó cégnél előfordul, hogy naponta akár több build is készül.

Agilis fejlesztés esetén a kódminőséget a refaktorálás / refaktorálás (refactoring) tevékenység hivatott növelni.

Az agilis szoftvertervezés szerves része a User Story / felhasználói történet / story point meghatározása.

Csapat vállalja a feladatokat és végzi a munkát és az ügyfél a csapattal együtt dolgozik.

- Önszerveződő, kereszt-funkcionális
méret: 2,3 – 20,25 (túl kicsi: módszertanok értelmét veszik; túl nagy: kommunikációs bajok)

Mit lehet agilisan csinálni?

- Szoftverfejlesztési módszertan
 - követelmények meghatározása
 - tervezés
 - kódolás
 - tesztelés ... stb
- Projektmenedzsment módszertan
 - projekt tervezés
 - projekt követés és vezérlés
 - változtatáskezelés

Agilis szerződések

Nem kerül minden rögzítésre, de valamit muszáj fixálni. Alapvetően más, de nem elég egy jó szerződés.

Sikeres szerződés támogatja:

- kollaborációt
- átláthatóságot/átlátszóságot (transparency)
- bizalom kialakulását és fenntartását

Scaled Agile Framework (TM)

Lean – Agilis, jól átgondolt, konzisztens, nagyvállalatban is használható (sima agilis nem)

3 szint:

- Vízió (Epic)
- Funkció (Higher level)
- Feature (User Story)

Lean

Vállalatszervezési rendszer, aminek célja, hogy a vállalat gazdaságosabban állítsa elő a termékeit, szolgáltatásait.

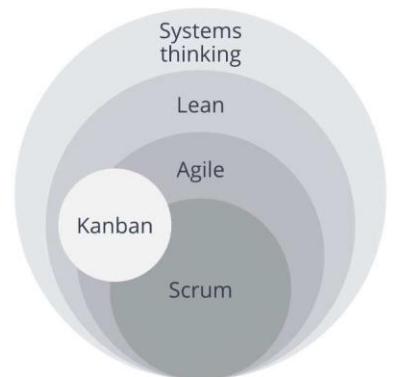
7 Lean alapelvek (összekapcsolták az agilitással)

1. Küszöböljük ki a veszteséget.
2. Hangsúlyozzuk a tanulást.
3. A lehető legkésőbb döntsünk
4. A lehető leggyorsabban adjuk át
5. Erősítsünk csapatokat, legyen hatalmunk
6. Építsük be az integritásra
7. Lássuk a teljes rendszert

Philosophies: Lean, Agile, etc.

Methodologies: Scrum, Kanban, XP, TPS, etc.

Tools: sprints, boards, tests, cohort analysis, etc.



Kanban: ráírják: To do, Doing, Done -> whiteboard -> fénykép
Dokumentációnak számít ha lefotózzák.



Agilitáshoz 3 alapvető igazság:

- Kezdetben nem tudni előre minden
- Garantált a változás
- Mindig több feladat lesz, mint pénz és idő

Nincs végső megoldás!

Folyamatfejlesztési modellek

Érettségi és képességi modellek:

Szoftvergyártás folyamatának elemeit és a szervezetet bizonyos érettségi szintekre sorolják kritériumok alapján.

Bizonyos kritériumok alapján vizsgálják a szervezetet és / vagy annak bizonyos vonatkozásait.

A vizsgált területek jellemzői szerint a szervezetet / vizsgált folyamatot bizonyos érettségi szintre sorolják.

Érettség

Először folyamatokra alkalmazták.

A szoftverfolyamat érettsége:

Annak mértéke, hogy egy folyamat mennyire definiált, tervezett, követett, mért, ellenőrzött, fejlődésre képes és hatékony.

Megmutathatja, hogy vajon a folyamat képes-e jó minőségű terméket előállítani úgy, hogy közben betartja a tervezett költség- és időkorlátokat.

Az érett szoftverfolyamat meghatározott, tervezett, követett, mért, ellenőrzött, hatékony és fejlődésre képes.

A szervezet érettsége: Egy szoftver gyártó szervezet annál érettebb (annál magasabb érettésgi szinten van) minél több **magas képességi szintű** (érett) folyamatot működtet.

Helyes szóhasználat, ami keveredhet:

Szervezet -> érettség

folyamat -> képesség

A két fogalom összefügg.

CMM = Capability Maturity Model (szervezet érettségét méri)

CMMI = Capability Maturity Model Integration (eredmény orientáltabb és bővebb)

Képességi-érettségi modellek

Jól meghatározott struktúrájuk van. A szoftverfejlesztés bizonyos elemeire koncentrálnak- Ellenőrzési mechanizmusokat fejlesztettek hozzájuk. (pl SCAMPI, Bootstrap...)

Lépcsős modellek (staged models)

Teljes szervezetet vizsgálják.

Foglalkoznak: Vezetési és műszaki folyamatokkal, alkalmazott technológiával, magával a szervezettel...

SW-CMM

Folytonos modellek (continous models)

Az egyes folyamatokra állapítanak meg képességi szinteket bizonyos jellemzők alapján.

modell alkalmazója dönti el, hogy milyen folyamat képességét szeretné vizsgálni

SPICE

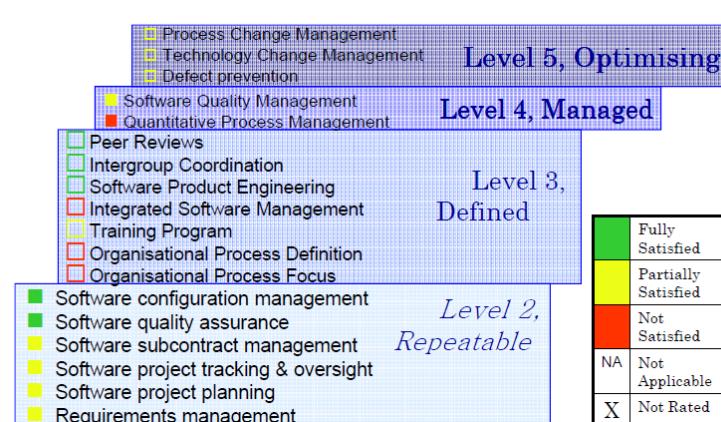
Kombinált, integrált modellek

Ötvözik a kétfélé modellt, hasznos elemeket kiválasztva

CMMI

CMM – Lépcsős modell (kevés)

5	Javításra felhasznált mennyiségi visszacsatolás	Folyamat változás menedzsment Technológia változás menedzsment Hibamegbízás	Termelőkénység és ciklus idő javítása	ellenőrzés
4	A hatékonyág, hatásosság, termelékenység és minőség mennyiségi biztosítása	Szoftver minőség menedzsment Mennyiségi folyamat menedzsment	Termelőkénység és a ciklusidő javulása	
3	Optimalizált módszerek dokumentáltak és projektenként használtak	Személk Csoportok közötti koordináció Szoftvertermék fejlesztés Integrált szoftver menedzsment Képzési terv Szervezeti szintű folyamatok meghatározása A folyamatok fejlesztése	A termékminőség lényeges javulása	
2	Hatókony módszerek léte	Konfigurációkezelés Minőségbiztosítás Beszállító kezelés Projekt követés & felügyelet Projekt tervezés Kötetelmények menedzsmentje	A projekt tervezés és vezetés megfelelő	
1	A projektek tipikusan átlépik az idő- és költségskeretet			
	10/31/2017			



SPICE – folytonos modell

Software Process Improvement and Capability dEtermination

Nemzetközi összefogás egy folyamatképességet vizsgáló szabvány létrehozására

Egy szervezet egyéni folyamataira vonatkozik.

Az ISO 15504-as szabvány:

- A szabvány a SPICE projekt eredménye
- Folyamatok képességének vizsgálata vonatkozó nemzetközi szabvány.
- Első kiadás: 1998.
- 2003 -2006 új kiadás és 2010 bővítés.

SPICE Képességi szintek:

- 5. Optimalizáló (optimising)
- 4. Jósolható (predictable)
- 3. Meghatározott/bevezetett (established)
- 2. Menedzselt (managed)
- 1. Végrehajtott (performed)
- 0. Nem teljes (incomplete)

Folyamatok: eredetileg szoftver életciklus folyamatok ISO 12207

(ábra) szerint, de

SPICE-ban folyamatokat ezekkel írják le:

- Folyamat azonosítója: Azonosítja a folyamatcsoportot és a folyamatot magát. A számozás magas szintű és második szintű folyamatokra utal. Tartalmazza még a folyamat nevének rövidítését.
- Folyamat neve: A folyamat legfőbb céljára utaló név. (pl Supplier Selection)
- Folyamat célja: A folyamat célját leíró mondat.
- Folyamat kimenetei: A folyamat kimenete, látható eredménye.
- Folyamatra vonatkozó megjegyzések: Opcionális

Minden képességi szinthez jó meghatározott gyakorlatok tartoznak. Ezek meglété ellenőrzik egy **audit** során.

Audithoz 3 elem szükséges

- Egy felmérési modell
- Egy felmérési módszer
- Egy vagy több kompetens auditor

Az ISO 15504 rendelkezik a szükségesekről!

Felmérés

Munkamódszer:

- Folyamatok kiválasztása
- Kérdőívek
- Megbeszélések
- Jelentés
- Regisztráció adatbázisba

Felmérés eredménye: egy érettségi profil, a kiválasztott folyamatokra.

A felmérés eredményének fontos része a folyamatjavítási terv.

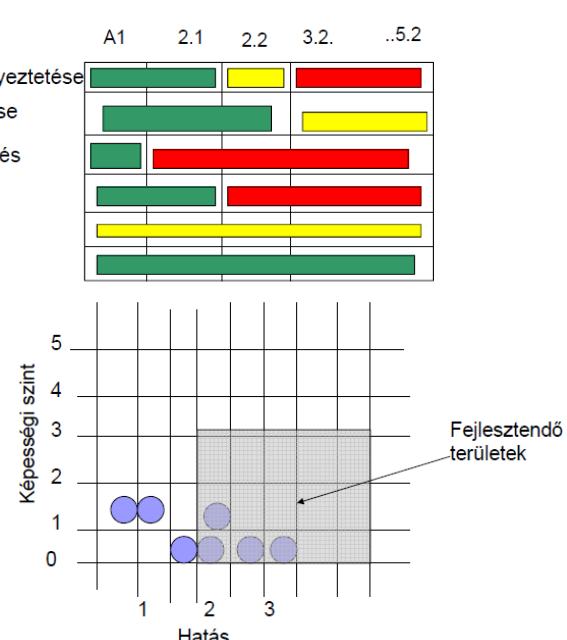
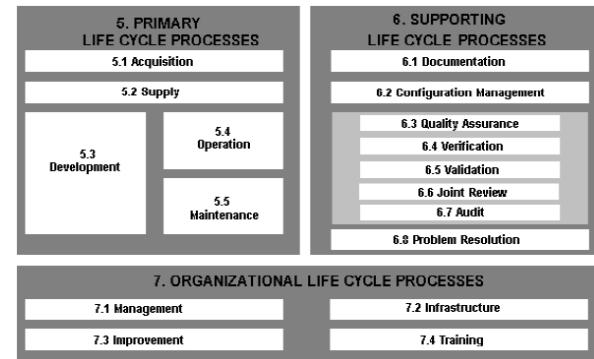
Értékelés:

A folyamat attribútuma mérhető jellemző. 0-100% közötti százalékos érték, ami megmutatja, hogy az adott attribútum mennyire teljesül.

Egy auditon a szervezet összes vizsgált folyamatára, a megcélzott képességi szinting, minden egyes folyamat-attribútumot értékelnek.

Ezeket aggregálva alakul ki az eredmény.

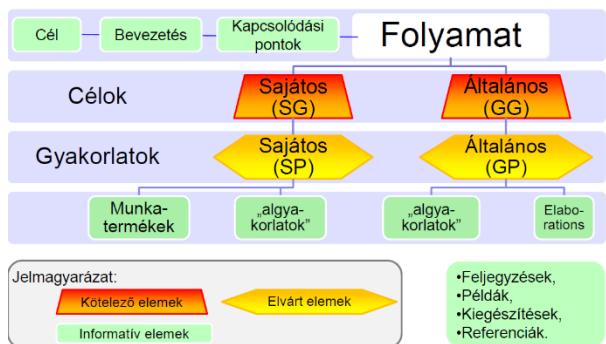
- Not achieved 0-15% - Nincs vagy kevés bizonyíték az adott attribútumról.
- Partially achieved 16-50% - Vannak bizonyítékok, de teljesítés nem egyenletes.
- Largely achieved 51-85% - Néhány munkatermék esetében a teljesítés nem egyenletes.
- Fully achieved 86-100% Bizonyíték van rá, hogy minden megfelelően működik.



CMMI – Integrált modell (most v1.3)

Capability Maturity Model Integration

Modell elemei: ábra->

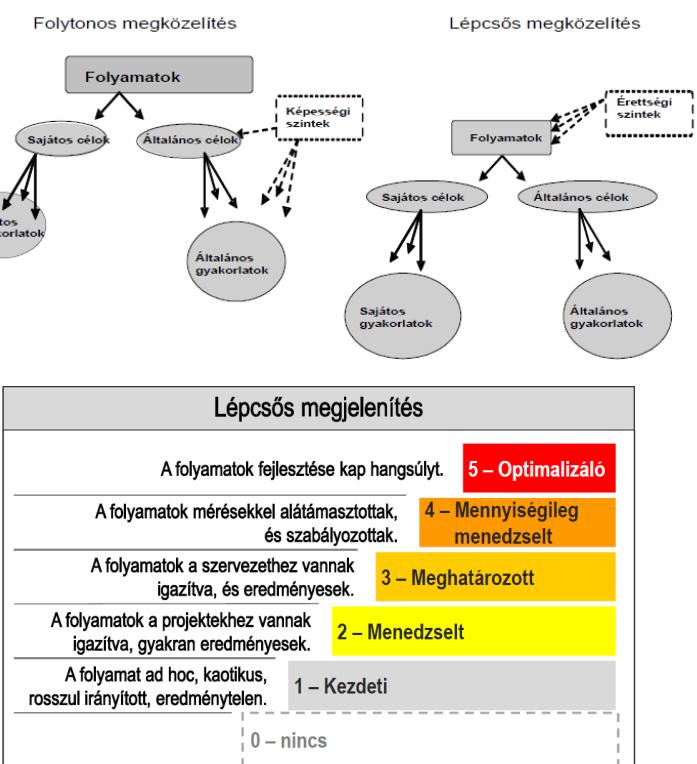


Képességi szint (folyamatra vonatkozó):

- 0. Incomplete
- 1. Performed
- 2. Managed
- 3. Defined

Érettségi szint (az egész szervezetre):

- 1. Initial
- 2. Managed
- 3. Defined
- 4. Quantitatively Managed
- 5. Optimising

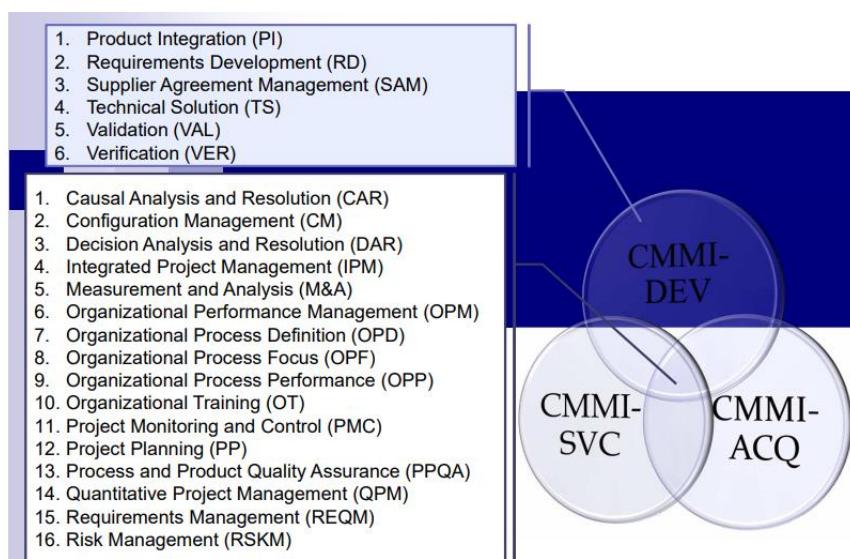


Konstelláció: sajátos területet ír le, ahol CMMI alkalmazható.

3 leggyakoribb:

- DEV – Development
- ACQ – Acquisition
- SVC – Services

CMMI folyamatok:



Minden esetben alkalmazandó folyamatok:

- CAR - Ok-kozat elemzés
- CM – Konfigurációmenedzsment
- DAR - Döntéselemzés és döntéshozatal
- IPM - Integrált projektmenedzsment
- MA - Mérés és elemzés
- OPD - Szerv. szintű folyamatok meghatározása
- OPF - Szerv. szintű folyamatszemlélet
- OPP - Szerv. szintű folyamat teljesítmény
- OT - Szerv. szintű képzés
- PMC - Projektkövetés és –vezérlés
- PP – Projekttervezés
- PPQA - Termék és folyamat minőségbiztosítás
- QPM - Mennyiségi (kvantitatív) projektmenedzsment
- REQM - Követelménymenedzsment
- RSKM - Kockázatmenedzsment.

Fejlesztésben (CMMI-DEV) alkalmazott folyamatok:

- PI - Termék integráció
- RD – Követelményfejlesztés
- SAM - Beszállító kezelés
- TS - Műszaki megoldás
- VAL – Validáció
- VER - Verifikáció

Minden esetben alkalmazott folyamatok (SVC) és Fejlesztésben alkalmazott (DEV)

Példa: Beszállítókezelés (SAM)

A szállítói megállapodás menedzsment célja, hogy a termékek beszerzése attól a szállítótól történjen, akivel létezik formális megállapodás.

Sajátos célok:

- SG 1 Szállítói megállapodás létrehozása
 - SP 1.1 Beszerzés típusának meghatározása
 - SP 1.2 Szállítók kiválasztása
 - SP 1.3 Szállítói megállapodás létrehozása
- SG 2 Szállítói megállapodás kielégítése
 - SP 2.1 Beszállítói szerződés végrehajtása
 - SP 2.2 Az értékelt (munka)termék elfogadása

Folyamatok, GG

Általános célok és az ezekhez kapcsolódó általános gyakorlatok- **ezek minden folyamat esetén azonosak!**

- GG1: Saját célok elérése (Achieve Specific Goals)

A folyamat létezik, végrehajtják, teljesíti sajátos céljait ->Az ilyen folyamat **CL1 (1-es érettségi szint)** -en van.

- GP1.1. Sajátos gyakorlatok végrehajtása (Perform specific practices)

Jelentése: végrehajtják a sajátos céljaihoz tartozó összes sajátos gyakorlatot.

- GG2: Egy menedzselt folyamat intézményesítése (Institutionalize a Managed Process)

- GP 2.1 Szervezeti politika (Establish an Organizational Policy)
- GP 2.2 A folyamat tervezése (Plan the Process)
- GP 2.3 Erőforrás biztosítás (Provide Resources)
- GP 2.4 Felelősségi körök kijelölése (Assign Responsibility)
- GP 2.5 Emberek képzése (Train People)
- GP 2.6 Konfiguráció menedzsment (Manage Configurations)
- GP 2.7 Az érintettek azonosítása és bevonása (Identify and Involve Relevant Stakeholders)
- GP 2.8 Folyamatkövetés és -vezérlés (Monitor and Control the Process)
- GP 2.9 Megfelelőség objektív vizsgálata (Objectively Evaluate Adherence)
- GP 2.10 Állapot szemlézése a felsőbb vezetőkkel (Review Status with Higher Level Management)

Az ilyen folyamat **2-es képességi szinten van (CL2)**.

			CL1	CL2	CL3
CM	Konfigurációmenedzsment	2			
MA	Mérés és elemzés	2			
PMC	Projektkövetés és -vezérlés	2			
PP	Projekttervezés	2			
PPQA	Folyamat és termék minőségbiztosítás	2			
REQM	Követelménymenedzsment	2			
SAM	Beszállítói kezelés	2			
DAR	Döntéselemzés és döntéshozatal	3			
IPM	Integrált projektmenedzsment	3			
OPD	Szerv. szintű folyamat meghatározása	3			
OPF	Szervezeti szintű folyamatszemlélet	3			
OT	Szervezeti szintű képzés	3			
PI	Termékintegráció	3			
RD	Követelményfejlesztés	3			
RSKM	Kockázatmenedzsment	3			
TS	Műszaki megoldás	3			
VAL	Végellenőrzés	3			
VER	Ellenőrzés	3			
OPP	Szervezeti szintű folyamatteljesítmény	4			
QPM	Mennyiségi projektmenedzsment	4			
CAR	Ok-kozat elemzés	5			
OPM	Szerv. teljesítmény menedzsment	5			

2-es
célprofil

3-as
célprofil

4-es profil

5-ös profil

Balla K.

- GG3: Meghatározott folyamat intézményesítése (Institutionalize a Defined Process)
 - GP 3.1 Meghatározott folyamat létrehozása (Establish a Defined Process)
 - GP 3.2 Javítási / fejlődési információk összegyűjtése (Collect Improvement Information)
- Az ilyen folyamat **3-as képességi szinten van (CL3)**

Folyamatcsoportok a CMMI-ben:

- Folyamatmenedzsment / Process Management
- Projektmenedzsment / Project Management
- Támogató / Supporting
- Fejlesztési (mérnöki) / Engineering (szoftvermérnöki folyamatokat/életciklus modellekkel valsz. itt írják le)

Lényeg:

CMMI-DEV folyamatai minden életciklus modellben szereplő szoftverfejlesztési/szoftvermérnöki folyamatok

Nem minden 1:1 a megfelelés (máshogy is lehet alkalmazni) pl:

- VER, VAL, PI = tesztelés
- TS = tervezés és kódolás

A CMMI a szoftverfejlesztési gyakorlatok viszonylag széles körben elfogadott értelmezését tartalmazza.

A CMMI verziói folyamatosa fejlődést mutatnak, követik a trendeket.

A CMMI megpróbál egyéb modellekből származó tudást integrálni.

A CMMI következetesen használta a benne szereplő folyamatokat az elmúlt 15-17 évben és továbbfejlesztette őket.

A továbbiakban a szoftverfejlesztéshez kapcsolódó folyamatok bemutatását a CMMI elvei alapján strukturáljuk.

A CMMI-nek való megfelelőség auditálása: SCAMPI módszer.

A CMMI felhasználók nehezen értik meg:

Technikai folyamatoktól különböző folyamatok tervezése és követése

CM - nem csak kódra

Adatmenedzsment – gyakran keverik a CM-tel.

IPM a maga teljességében (gyakran vita tárgya, hogy az IPM a PP és PMC „összege”, ezért nem is kell rá külön folyamatleírás...)

GP-k, különösen a PP és PMC tervezése, a „mérés mérése”, a „CM konfiguráciomenedzsmentje”, érdekelt felek...

TMMi: CMMi modellen alapuló tesztelési folyamatok érettségére vonatkozó modell

Egyéb folyamatfejlesztési modellek

PSP

CMM elvein alapuló oktatás(tanfolyam)!

A szoftverfejlesztőket egyenként segíti teljesítményük javításában, a fegyelmezett munkavégzés bevezetésében

Folyamat részei:

- Egyéni mérés
- Egyéni tervezés
- Egyéni minőség
- Kalibrálás

6+1 fázis:

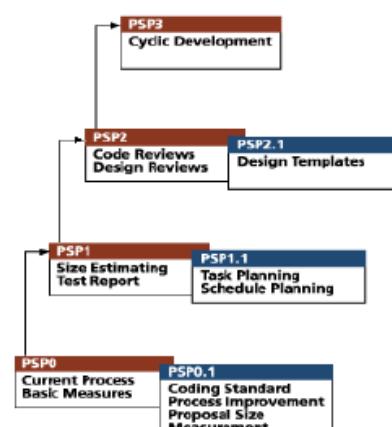
1. planning
2. designing
3. coding
4. compiling
5. testing
6. post morten

(+1) a programozó munkájának értékelése – számos formátum támogatásával

CMMi és PSP

CMMi:

- szervezetre vonatkozik
- keret, amiben jó minőségű szoftverfejlesztés végezhető
- keret a folyamatok javítására
- feltételezi, hogy a fejlesztők hatékony módszereket ismernek, alkalmaznak és javítanak



PSP:

- egyénre vonatkozik
- feltételezi, hogy van egy rendezett és vezetett keret, amiben dolgozni kell
- megvan az infrastruktúra a megfelelő szoftverfejlesztési folyamatok támogatására

TSP

Ahhoz, hogy a fejlesztők hatékonyan alkalmazzák a PSP-t, szükséges:

- saját munkamódszerük és a csapat munkamódszerének összekapcsolása
- irányítást és támogatást kapniuk a fegyelmezett munkavégzésben

Szükség van magas szintű csapatokra, amelyek jó szoftvert gyártanak, betartják a határidőket, nem lépik túl a költségkeretet és munkájukat folyamatosan javítják.

TSP képzés előtt PSP képzésen részt kell venniük a fejlesztőknek.

A TSP-t alkalmazó csapatok:

3-20 fős, szoftverfejlesztéssel foglalkozó, vegyes csapatok

Fejlesztők saját munkamódszerüket a csapatéval össze kell kapcsolni és muszáj irányítást és támogatást kapniuk.

IDEAL

Szervezeti folyamatfejlesztési modell, amely segít a fejlesztési

igényt felismerni, változást folyamatosan fenntartani.

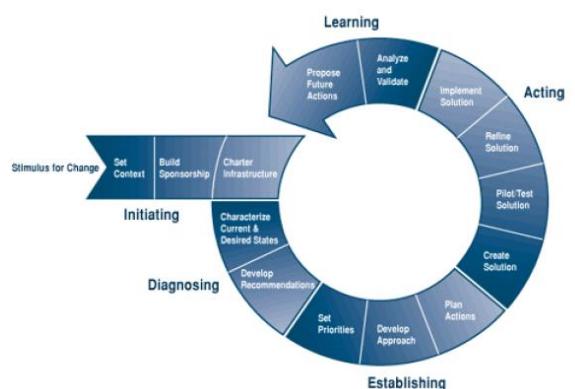
Initiating, Diagnosing, Establishing, Acting, Learning

Agilis folyamatfejlesztés

Agilisan dolgozó szervezetekben végzett folyamatfejlesztés.

A folyamatfejlesztési tevékenységek agilis módon való végrehajtása. (pl. kevesebb dokumentációval)

Tapasztalat szükséges, de érdekes!



Követelmények kezelése

Követelmény Definíció:

- 1. Olyan feltétel vagy képesség, amely a felhasználó számára azért szükséges, hogy megoldjon egy problémát vagy elérjen egy célt.
- 2. Ezen feltételnek vagy képességek a rendszer vagy rendszer komponens által is megvalósíthatónak kell lennie úgy, hogy közben a szerződés, szabvány, specifikáció és egyéb formális dokumentumban támasztott követelményeknek is megfeleljen. (Klasszikus 1990-ből)
- **Modernebb definíció:**
- 1. Egy érdekelt fél által felismert igény.
- 2. Olyan képesség vagy tulajdonság, amellyel a rendszernek rendelkeznie kell.
- 3. Egy igény, képesség vagy tulajdonság dokumentált leírása.

Problémák:

Vevők nem tudják mit akarnak vagy nem tudják elmagyarázni -> Kommunikációs problémák/ félreértesek a vevők, fejlesztők és menedzsment között.

A szoftver működési elemeit „magától értetődnek” tekintik.

Időkövetelmény -> teljesíthetetlen határidők majdnem minden.

Követelmények változnak a projekt során. A felhasználók nincsenek tudatában a változások hatásának.

Hírhedt hibák követelmények pontatlansága miatt:

Az Ariana-5 első tesztrepülése-> rakéta felrobbantotta magát 37 mp után; Oka: 64->16 bit konverzió

Mars Climate Orbiter -> eltűnt a marsjáró 41 hét után; Oka: mértékegység (angol – metrikus)

nem voltak pontosak a követelmények és nem volt megfelelő integrációs teszt!

Miért fontos a pontos és világos követelmények?

Rendszer határát kijelöljük és a rendszer azt csinálja, amit kell.

Be tudjuk bizonyítani, hogy elvégeztük a feladatot.

Rendet tudunk tartani a követelmények között -> szükséges dokumentáció

Követelmény leíró dokumentumok segítenek a

- Tervezés és implementációban
- Teszt esetek, forgatókönyvek létrehozása
- Felhasználói dokumentáció létrehozása

A szoftvermérnökök **segítenek a követelmények megfogalmazásában**.

Egyes követelmények a felhasználótól jönnek: **ilyenkor le kell lefordítani** (felhasználói) őket termék/rendszer követelményekre.

Néha **fel** is kell követelményt **találni**. (nincs rá kérés) – Gondolunk a Facebookra!

A követelményeket a kódban is pontosan be kell tudni azonosítani. Ennek egyik módja, hogy a kódrészletekbe kommentként beírjuk a vonatkozó követelményeket.

Követelmény fejlesztés (RE – Requirements engineering)

Alaptevékenységek:

- Meghatározás, egyeztetés, felmérés
- Dokumentálás
- Validáció, egyeztetés
- Követelmények menedzsmentje

Idő és erőforrások szükségesek.

Lényeges eleme a szoftverfejlesztésnek.

Célok:

- A helytálló, megfelelő követelmények megismerése, az érdekelt felek egyetértésének elérése ezekre a követelményekre vonatkozóan, a követelmények szabványos dokumentálása, a követelmények folyamatos menedzselése.
- Az érdekelt felek óhajainak megértése és dokumentálása
- Követelmények meghatározása és menedzselése, hogy átadáskor csökkenjen a kockázata annak, hogy nem felel meg az érdekelt feleke elvárásainak és szükségleteinek, szóval megfelelő legyen.

CMMI-ben

ML2: Requirements management /Követelménymenedzsment

ML3: Requirements development / követelményfejlesztés

Követelménymenedzsment

Célja a projekt termékeihez és termék komponenseihez szükséges követelmények menedzselése, és a követelmények és a projekt tervezési és munkatermékek között esetleg fellépő ellentmondások azonosítása.

- **SG1 Követelmények menedzselése**

- SP 1.1 Követelmények megértésének elérése
- SP 1.2 A követelményhez való elkötelezettség elérése
- SP 1.3 Követelményváltozás menedzselése
- SP 1.4 Kétirányú követhetőség fenntartása
- SP 1.5 A projekt munka és követelmények közötti ellentmondások azonosítása

Ezek a tevékenységek a **CMMI 2-es érettségi szintjén szükségesek!** Ez azt jelenti, hogy alapvetően fontosak!
Követelményfejlesztés

Célja a vevői, termék, és termék-komponens követelmények felmérése, elemzése és dokumentálása.

- **SG 1 Vevői követelmények fejlesztése**

- SP 1.1 Szükségletek felderítése
- SP 1.2 Érdekeltek felek igényeinek vevői követelményekké alakítása

- **SG 2 Termékkövetelmények fejlesztése**

- SP 2.1 Termék és termék-komponens követelmények meghatározása
- SP 2.2 Termék-komponens követelmények allokálása
- SP 2.3 Interfész követelmények azonosítása

- **SG 3 Követelmények elemzése és jóvahagyása**

- SP 3.1 Működési elképzelések és forgatókönyvek meghatározása
- SP 3.2 Az igényelt funkcionális és minőségi jellemzők definiálása
- SP 3.3 Követelmények elemzése
- SP 3.4 Követelmények elemzése egyensúlyi állapot eléréséhez
- SP 3.5 Követelmények validálása

Ezek a tevékenységek a **CMMI 3-as érettségi szintjén szükségesek!** Mélyebb szakmai tudást feltételeznek!

Követelmények forrásai

Alapvetően:

- felhasználó
- hasonló rendszerek
- általános tudás a témaival kapcsolatban

Forrásoknál felszokás írni: Név, beosztás, kontakt adatok, relevancia, tapasztalat, célok.

Hozzunk létre szótárt, amely tartalmazza a követelményekbe megfogalmazott alapfogalmakat!

Amiket vevők nem szoktak specifikálni: Szabványok, Üzleti alapelvek, tervezési döntések és törvények

Követelmények egyeztetése

Alapvető a felhasználóval kommunikálni.

Egyeztetési technikák:

- felmérési technikák – pl interjúk, kérdőívek
- kreatív technikák – pl brainstorming, nézőpont változtatás
- dokumentum központú technikák – pl rendszer-régészeti, követelmények újrafelhasználása
- megfigyelési technikák – pl helyszíni megfigyelés, tanonckodás
- támogató technikák – pl mind mapping, audio-video felvételek, use case modellezés...

Cél, hogy egyetértés szülessen a követelményekről a különböző felek között.

Feladatok: Konfliktus

- azonosítása
- elemzése
- megoldása
- megoldás dokumentálása -> tapasztalatok miatt

Az egyeztetett követelmények elemzése abból a célból, hogy megértsük és dokumentáljuk őket – ennek a szinonimája a követelményfejlesztés.

Gondolkodjunk! – megbeszélések, brainstorming

Csoportosítsunk és priorizálunk követelményeket.

Követelmények típusai

- Funktionális
- Nem funkcionális vagy minőségi

Követelmény-típusnak tekinthetjük még:

- Üzleti követelmények
- Felhasználói követelmények
- Termék követelmények
- Interfész követelmények

Funkcionális követelmények

Amit a rendszer csinál.

Funkcionális követelmény: olyan követelmény, amely a szoftverrel szemben támasztott funkcionális elvárást írja le.

Funkcionális követelményspecifikációt **az általános hallgatószámára csinálják**. Az olvasónak meg kell értenie a rendszert, de semmilyen sajátos vagy technikai tudásra nincs szükség ehhez.

Nem azonos az üzleti követelményekkel.

Tartalmaznia kell:

- A rendszer bemenő adatainak leírását.
- Az egyes képernyők által megvalósított műveletek leírását
- A munkafolyamatok leírását
- A kimenő adatokat és jelentéseinek leírását
- Annak leírását, hogy ki vihet be adatot a rendszerbe
- Annak leírását, hogy mi felel meg a érvényes törvénynek.

Nem-funkcionális követelmények

Ahogyan a rendszer csinálja.

Expliciten dokumentálni kell, főleg: Teljesítmény, biztonság, megbízhatóság, használhatóság, karbantartás, hordozhatóság.

Pl: A rendszernek maximum 10000 felhasználót kell egyszerre kezelnie. A válaszidőnek minden 2 sec alatt kell lennie. A rendszernek PC-n, tableten és Androidot használó okostelefonon is működnie kell. A rendszernek a hét minden napján, 0-24 óra között működnie kell.

Felhasználói követelmények

Magas szintűek a kezdeti bizonytalanság ellenére. (megfogalmazási gondok)

Mit akarunk, nem pedig hogyan oldjuk meg.

Kerüljük, hogy tervezési vagy implementációs megoldások követelményként kerüljenek leírásra!

Termék követelmények

Segítenek megérteni, hogy új elemeket felhasználni.

Például ezeket tartalmazzák:

- Cél
- Központi elemek
- Felhasználói adatfolyam
- Analitika – számok, amik megmutatják mennyire jó a termék (pl: látogatók száma oldalanként)
- Jövőbeli Kiegészítések

Korlátozások

A korlátozás olyan követelmény, amely leszűkíti a megoldási lehetőségeket azon belül, ami ahhoz szükséges, hogy a rendszer a számára leírt funkcionális vagy nem funkcionális elvárásokat teljesítse.

Ezeket nem lehet befolyásolni.

Követelmények modellezése

Fogalmi modellek szoktak leírni őket – grafikai elemekkel, természetes nyelven vagy kombinálva

Cél modellek: érdekelt fél szempontjait írja le. Lebontás hasznos.

Use case-ek: Segítenek megvizsgálni a felhasználó szemszögét.

3 nézőpont:

- Adatközpontú követelménymodellezés: Tipikusan entity relationship diagrams
- Viselkedés központú követelménymodellezés: Rendszer állapotai és az ezeket befolyásoló események. UML állapotdiagrammok
- Funkcionalitás központú követelménymodellezés: Az input adatok outputtá alakításával foglalkozik. UML kontextus diagramok

Ellenőrizzük a követelményeket és modelleket is.

Alapelemek:

- Adatok: Hangsúly rendszer által felhasznált és létrehozott információkon van.
- Akciók: Van inputjuk és outputjuk és lebonthatók
- Eszközök: minden rendszernek van portja -> I/O események.
- Események: olyan rendszerszintű input vagy output, amely a porton jelenik meg
 - Diszkrét események: adott ideig kapcsolódik hozzájuk
- Vezérfonalak: leginkább tesztelésben használatos

Dönteni kell melyiket alkalmazzuk, de NINCS egyetlen jó megoldás.

Követelmények prioritálása

Követelményeket sorrendbe állítjuk, különböző időpillanatokban, tevékenységek során, kritériumok szerint.

Sorrend -> valamit előnyben részesít. pl:

- A prioritálás céljának és korlátainak meghatározása
- A prioritálás kritériumainak meghatározása
- A releváns érdekelt felek azonosítása
- A prioritálandó elemek kiválasztása

Követelmények dokumentálása

Kiderüljön belőle a rendszer célja, alapvető rendeltetése, szoftver típusa, becsült idő és költségek.

Szoftverfejlesztés több fázisában fontos.

Mitől jó?

- Egyértelmű
- Konziszens
- Szerkezete világos
- Módosítható és kiterjeszthető
- Teljes
- Követhető, hogy melyik követelmény hogyan fejlődik, bomlik részekre... stb

Követelmény validálása:

A követelmény validálás során annak (vég)ellenőrzése történik, hogy a követelmények teljesítik-e a velük szemben megfogalmazott minőségi kritériumokat, azzal a céllal, hogy a hibákat – a követelményfejlesztés folyamatában minél előbb megtalálják és kijavítsák.

Erre módszerek:

- Kommentelés (szakértői)
- Inspekción
- Walkthrough

A követelmények menedzsmentje jelenti a követelmények gondos kezelését, kiemelten figyelve az alábbiakra:

- Kétirányú követhetőség
- Követelmények változásának kezelése

Kétirányú követhetőség

A követelmények közötti oda-vissza, valamint horizontális és vertikális kapcsolatokat hivatott megjeleníteni

- **oda-vissza (horizontális):**
 - Vevői követelmény -> termék követelmény -> design elemkód részlet -> teszt eset -> a működő rendszer eleme -> felhasználói dokumentáció eleme ... stb. ÉS visszafele is...
 - Ez a horizontális kétirányú követhetőség
- **vertikális:** magas szintű követelmény lebontása horizontálisan követhetőkre

Követelmények változásának kezelése

Lehetővé teszi, hogy elkerülhetetlen módosítások miatt ne csússzon ki a kezünk ből a szoftver követelményrendszere

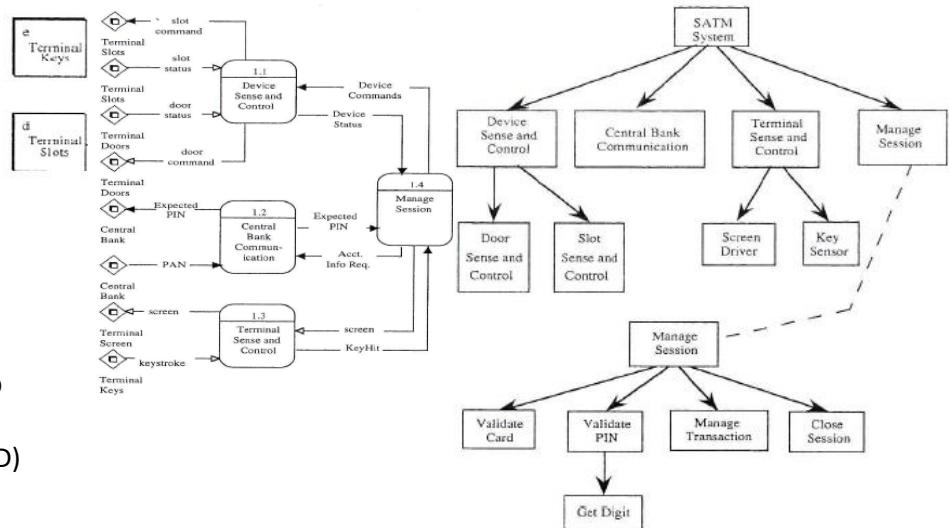
Változások: új, törlés, módosítás

Hogy kell: Verziókezelés -> alapverzió!

Hatáselemzés és dokumentálás.

Követelmények modellezése:

- Kontextus diagram



- Adatfolyam diagram (bal)
- Funkcionális dekompozíció (jobb ábra)
- Entity/ relationship diagram (ERD)
- Működési modell
- Use-Case modell
- Szekvencia diagram

Követelmények agilis környezetben

Itt is fontos a változások kezelése!

Alapvetően user storykat használ.

Felhasználói történet (user story):

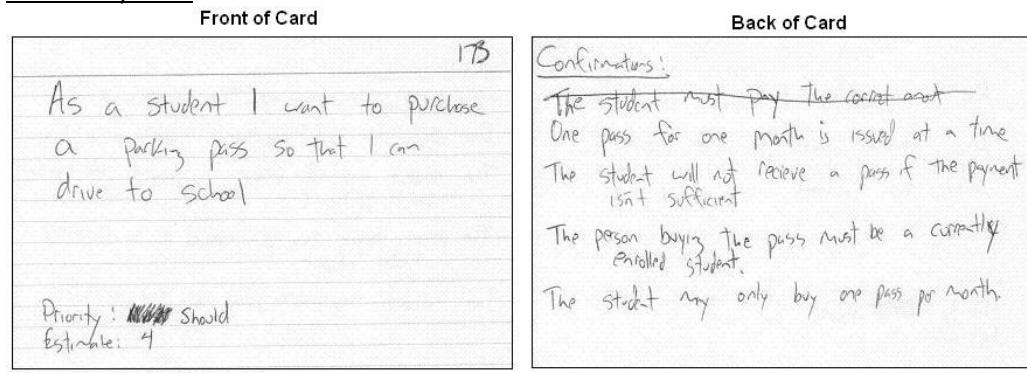
Magas szintű felhasználói, vagy üzleti követelmény, amelyet általában az agilis szoftverfejlesztés során használnak. Jellemzően egy, vagy több, hétköznapi, vagy üzleti nyelven megfogalmazott mondatot tartalmaz, amely leírja, hogy a felhasználónak milyen funkcionálisra van szüksége, vagy bármilyen egyéb nem-funkcionális követelményt fogalmaz meg, továbbá tartalmazza az átvételi kritériumot is.

saját pontatlan megfogalmazás: Leírja, hogy a felhasználónak milyen funkciókra van szüksége mondatokban. Egységes munkát ír le.

Segít a csapaton belül megérteni a követelményt

Ki mit akar csinálni és miért?

User Story Card:



Copyright 2005-2009 Scott W. Ambler

Agilis környezetben is szükséges a fegyelmezett változáskezelés!

Jó követelményeket írni, melyek alapján jó szoftvert lehet tervezni és fejleszteni – **nem könnyű**.

Fontos, hogy korábbi tapasztalatokból tanulunk!

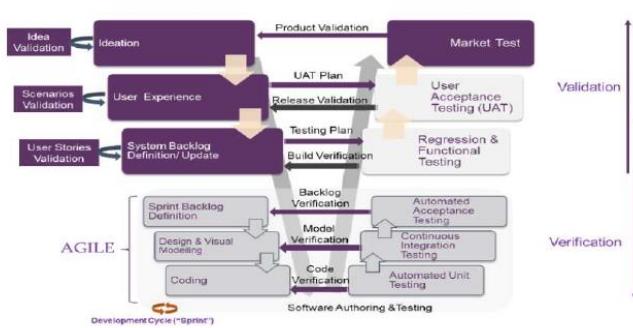


Figure 3-2. The CloudTeams methodology expanded with an Agile Software Development methodology

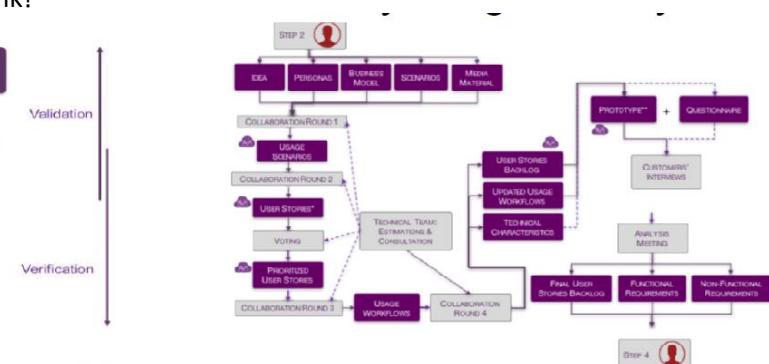


Figure 5-7. System backlog definition and User Stories Validation

Tervezés, implementáció

Tervezés / Design

Definíciók

- Az a folyamat, ami során a követelményeket kiindulási alapnak használva egyre pontosabban és részletesebben megértjük a rendszert, annak minden elemével együtt.

A megértés formalizálása különböző nézőpontokból történik:

- Funkcionalitás
- Funkciók
- Adatok
- Kommunikáció

A formalizálás különböző tervezési modellek felhasználásával történik.

- A követelményeket lefordítjuk Funkcionális elemekre és Szoftver struktúrára.
- **Design:** mind „az architekturális emelek, komponensek, interfések és egyéb rendszer elemek definíciója”, mind „ezen folyamatok eredménye”
- Ha folyamatként vizsgáljuk, a szoftvertervezés a szoftverfejlesztési életciklus azon tevékenysége, amelynek során a követelményeket elemezzük abból a célból, hogy elkészítsük a szoftver belső struktúrájának leírását, amely az implementáció alapjául fog szolgálni.

A tervezés folyamata

A tervezés végére a teljes rendszer koherens, összefüggő struktúráját le kell írni!

Nem kódolás!

Tervezés részletessége:

- Szoftver architektúra terv: a szoftver legfelső szintű struktúráját írja le és azonosítja a komponenseket (Fogalmi terv, Conceptual, High level design)
- A szoftver részletes terv: a komponenseket olyan részletezzéggel írja le, amely lehetővé teszi, hogy implementálni lehessen őket (Részletes terv, detailed design)

Statikus és dinamikus rendszer nézetet írunk le.

Egy modell több életciklus fázisban használható, tehát nincsenek csak tervezésre használhatók!

Más modellek más elemeire koncentrálnak a rendszernek. Együtt jobb rálátást adnak.

Szoftvertervezési alapelvek és fontos elemek

Olyan alapelemek, elvek, amelyeket többféle szoftvertervezési megközelítésben is elfogadnak és alkalmaznak.

Volt már szó róla. (UML2 OO Design Principles)

Néhány **minőségi követelmény**, amelyek minden szoftverre érvényesek: teljesítmény, biztonság, megbízhatóság, használhatóság. Ezeket tartuk szem előtt!

Jól ki kell találni, hogy kell felbontani, szervezni, csomagolni a szoftver komponenseket!

Környezetet is figyelembe kell venni.

Architektúrák, döntések, tervezési minták

Egy szoftver architektúra azoknak a struktúráknak az összessége, amelyekkel foglalkozni kell / tekintetbe kell venni őket a rendszer fejlesztésekor. Magába foglal szoftver elemeket, a közöttük levő kapcsolatokat, valamint az elemek és kapcsolatok tulajdonságait. Az architektúra –stílus elemtípusoknak és relációtípusoknak egy speciális halmaza, az alkalmazásukra vonatkozó korlátozásokkal együtt.

Architektúra stílusok: a szoftver szerkezetének magas szintű leírását adja.

Példák:

- Általános struktúrák (pl. rétegek, csőrendszerek és szűrők...)
- Elosztott rendszerek (pl. kliens-szerver, háromrétegű...)
- Interaktív rendszerek (pl. Model-View-Controller, PresentationAbstraction-Control)
- Egyéb (pl. batch, interpreter, folyamat vezérlő, szabály-alapú).

Tervezási minták: Egy minta „egy általános megoldás egy adott kontextusban megjelenő általános problémára”.

Az architektúra stílusok tekinthetők a szoftver-szerkezet magas szintű szervezési mintáinak.

Példák:

- Létrehozási minták (pl. builder, factory, prototype, singleton)
- Strukturális minták (pl. adapter, bridge, composite, decorator, façade, flyweight, proxy)
- Viselkedési minták (pl. command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor)

Tervezési döntések:

A tervezés kreatív folyamat.

Tervezőknek sok döntést kell meghozniuk, amely befolyásolja a szoftver struktúráját és a teljes szoftverfejlesztési folyamatot.

A teljes tervezési folyamatot tulajdonképpen tekinthetjük döntések sorozatának is.

A döntések során mérlegelni kell a különböző minőségi attribútumokat és vevői igényeket, egyensúlyra törekedve. Vannak architekturális döntések is. (Hogy dokumentáljuk, Van hozzá minta)

Újrafelhasználás:

Programcsaládok és keretrendszerek:

- Az újrafelhasználást támogatja, ha eleve programcsaládokat tervezünk; ezeket terméktípusoknak is nevezik (software product lines).
- Ilyenkor meg kell határozni az elemek közötti kommunikációt, és arra kell törekedni, hogy az elemek önállóak, jól dokumentáltak legyenek.
- Az OO fejlesztésben alapvető elgondolás, hogy „részben kész” keretrendszereket fejlesztenek, amelyekhez később új elemek hozzáadhatók (pl. plug-in-ek)

Felhasználói interfész (UI) tervezése

Alapelvek:

- Tanulhatóság. Könnyen megtanulható legyen, ösztönözve a felhasználót arra, hogy minél előbb kezdje el használni.
- Ismerős a felhasználónak. A felhasználó számára ismerős fogalmakat használjon
- Konzisztencia. Az interfész tegye lehetővé, hogy hasonló funkciókat hasonlóan lehessen elérni / indítani.
- Minimális meglepetés. A szoftver viselkedése ne lepje meg a felhasználót.
- Visszaállíthatóság. Hiba után az interfész tegye lehetővé az újraindítást.
- Felhasználó támogatása. Az interfész adjon érdemi visszajelzést a felhasználói hibákra, és nyújtson segítséget a használatban.
- Felhasználói sokféleség. A UI tegye lehetővé, hogy a rendszert többféle felhasználó használhassa (pl. vakok, gyengénlátók, színtévesztők stb...). 21 2017

Szoftvertervezés a CMMI és az Automotive SPICE modellekben

CMMI

Nincs egyetlen folyamat Design néven.

Követelményfejlesztés (RD, ML3):

Célja a vevői, termék, és termék-komponens követelmények felmérése, elemzése és dokumentálása.

- SG 1 Vevői követelmények fejlesztése
 - SP 1.1 Szükségletek felderítése
 - SP 1.2 Érdekkelt felek igényeinek vevői követelményekké alakítása
- SG 2 Termékkövetelmények fejlesztése
 - SP 2.1 Termék és termék-komponens követelmények meghatározása
 - SP 2.2 Termék-komponens követelmények allokálása
 - SP 2.3 Interfész követelmények azonosítása
- SG 3 Követelmények elemzése és jóváhagyása
 - SP 3.1 Működési elképzélések és forgatókönyvek meghatározása
 - SP 3.2 Az igényelt funkcionálitás és minőségi jellemzők definiálása
 - SP 3.3 Követelmények elemzése
 - SP 3.4 Követelmények elemzése egyensúlyi állapot eléréséhez
 - SP 3.5 Követelmények validálása

Műszaki megoldás (TS, ML3):

A műszaki megoldás célja a követelmények szerinti megoldások tervezése, fejlesztése és megvalósítása.

- SG 1 Termék-komponens megoldások kiválasztása
 - SP 1.1 Alternatívák és kiválasztási kritériumok kidolgozása
 - SP 1.2 Termék-komponens megoldás kiválasztása
- SG 2 A (műszaki) terv fejlesztése
 - SP 2.1 A termék vagy termék-komponens tervezése
 - SP 2.2 Technikai adatcsomag meghatározása
 - SP 2.3 Interfész-használati kritériumok megtervezése
 - SP 2.4 Elemzés: készítés, vásárlás vagy újrafelhasználás?

- SG 3 Termék fejlesztési modell implementálása
 - SP 3.1 Fejlesztési modell implementálása
 - SP 3.2 Terméktámogatási dokumentáció elkészítése

Automotive SPICE

Nagyon fontos beágyazott rendszereknél!

A rendszert különböző szintű architekturális elemekre bontják.

A szoftvert is felbontják különböző szintű architekturális elemekre, míg eljutnak a legalacsonyabb szintű elemhez; ezt szoftver komponensnek nevezik.

Design dokumentálása

Magas szintű és részletes terv.

Modellek, leírások és döntések alapjának leírása.

Verziókezelés a design dokumentumoknál is fontos, elengedhetetlen!

Segít közös nevezőre hozni a különböző érdekeltek felek értelmezését a rendszerről. (Segít értelmezni)

Teljes design-t a műszaki adatcsomag tartalmazza.

Legyen szabványos, érthető, konzisztens és nem redundáns!

A jó szoftvertervező **több lehetőséget, alternatívát is megvizsgál**, értékel.

Implementáció / kódolás

Nem csak kódolás!

Újrafelhasználás, konfiguráció menedzsment, „Host-target” fejlesztés még!

Kódolási szabványok alkalmazása

Kódolás, technikai/műszaki dokumentáció elkészítése!

Struktúra, kommentek, változók nevei...

Programozási gyakorlatok ismerete fontos.

Kerüljük más hibáit → korábbi tapasztalatok felhasználása.

Kétirányú követhetőségnek kell lennie a követelmények – design – kód – teszt eset között!

Becslés a szoftvertervezés során

A becslés segít megérteni, hogy mi is fog történni, illetve „milyen és mekkora” lesz a rendszer.

Ezekre becslünk:

- Ráfordítás
- Költség
- Méret

Nem könnyű, tapasztalat elengedhetetlen. -> PROBE módszer (PSP) IMSC

Funkciópont számolás:

Kifejlesztésének célja: különböző technológiákkal történő szoftverfejlesztések hatékonyságának összehasonlítása.

Később rájöttek, hogy a módszer jól alkalmazható a specifikáció alapján történő becsléskor. Van rá két módszer.

IFPUG módszer

Nincsenek pontos definíciók.

Legtöbbet használt, sok a tapasztalat nemzetközileg is.

Adatfeldolgozó rendszerekben jó. Valós idejű rendszerekben rossz.

Csak alkalmazás – típusú szoftverekre

MKKI módszer

Lépései:

- A számlási nézőpont, cél és típus meghatározása
- A számlás korlátjainak meghatározása
- A logikai tranzakciók azonosítása
- Az entitások típusának meghatározása és besorolása
- A bemenő adat-elemek típusának, a meghivatkozott entitás típusoknak és a kimenő adat-elemek típusának megszámolása
- A funkcionális méret kiszámolása
- $FPI = Wi \times \Sigma Ni + We \times \Sigma Ne + Wo \times \Sigma No$, ahol FPI = Function Point Index Wi, We és Wo az Ni, Ne és No értékek súlyozott átlaga, éspedig: Wi = 0.58, We = 1.66 és Wo = 0.26.
- A projekt ráfordításának meghatározása
- A termelékenység és egyéb mutatók kiszámolása

IFPUG továbbfejlesztése. Életciklus korai fázisában alkalmazható.

Csak alkalmazás – típusú szoftverekre

Cosmic módszer

Egyszerű és egyértelmű.

Méretezési szabályok:

Funkcionális processz mérete: min 2, nincs max

Egy szoftverelem mérete a funkcionális proceszei méretek összege.

Tervezés és implementáció agilis környezetben

A megoldás korai bemutatása/kipróbálása!

Funkciók, funkciócsoporthoz, release bemutatása.

Ha később az eredeti fejlesztőkön kívül más fogja továbbfejleszteni a rendszert, kiemelten fontos a tervezési dokumentáció átadása is.

Felhasználó a középpontban.

Agilis Becslés: Ugyanaz a cél, a technika különbözik.

Hagyományos tervezés	Agilis tervezés
Sorozatban történő munkafolyamat; checkout, egyszerre egy valaki tervez	Párhuzamos folyamat, egyszerre többen is terveznek
A feladatokat hetek, hónapok alatt végzik el	A tervezési feladatokat percek, órák, napok alatt végzik el
E-mail-ek, nyomtatás, meetingek	SMS, videohívás...
Irásasztalnál dolgozó, fix csapat	A csapattagok bárhol lehetnek
A tervezési adatokat a tervezők magunknál tartják; a tervezők "csak" terveznek	A tervezési adatok mindenki számára elérhetők, a csapattagok széleskörű tudással rendelkeznek
Az innovációs igények miatt megpróbálnak eszközöket használni	Az eszközök felgyorsítják a munkát és az innovációt
A menedzsment általában hónap végén, már lejárt adatokat kap	A menedzsment folyamatosan tájékozódik, valós időben

Agilis Design elemek:

Magas szintről indulunk, majd végül eljutunk a kódig.

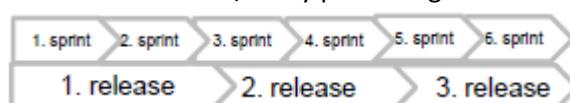
Közben elképzeljük a rendszer architektúráját, átgondoljuk a modelt, írnak egy egyszerű tesztesetet... stb

Az agilis szoftvertervezés szerves része a User Story / felhasználói történet / story point meghatározása

USER STORY meghatározása:

1-2 Sprint után release.

A követelményeket addig be kell fagyasztani!



A következő sprintre csak kis változtatások, letisztult ötleteket megengedettek.

NB.: a szoftvertervezés (design) és a projekttervezés egyszerre történik!

A Sprint végén gyártásra kész funkciók kerülnek bele és a fejlesztőcsapat mutatja be.

Ez egy „Követelmény”, amit az ügyféllel együtt csinál a fejlesztő.

JUST IN TIME – mindenkor megfelelő időben kell használni.

Sok fog keletkezni érdemes virtuálisan tárolni, de fizikainak is van előnye.

INVEST -> Független, egyeztethető, értékes, becsülhető, kicsi, tesztelhető (user story minősége ettől függ)

User story mapping:

- Időrendben sorrendben nézzük a releaseket és a benne történő folyamatokat.
- Letisztult kép további részletekért.

Burn-down chart: ábra oldalt.

Az agilis design és projekttervezés

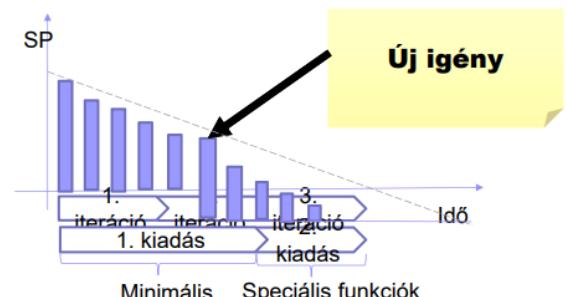
Számos egyéb változat van a burn up/down- chart-ra.

Reális terv kell.

Ne számítsunk csodára.

Definition of Done:

Elemzett, tesztelt, implementált.



- Egyeztetett / mindenki által elfogadott listája a termék inkrementum elkészítéséhez elengedhetetlenül szükséges tevékenységeknek
- A csapat megegyezésre jut a DoD-t illetően, és ki is függesztik ezt a teremben egy jól látható helyre. Egy sor követelménynek teljesülni kell ahhoz, hogy a termék adott inkrementumát (általában egy user story-t) elkészültnek tekintenek.
- Ha a kritériumok nem teljesülnek a sprint végére, az adott tevékenységeket nem számolják bele a csapat sebességébe.

Vegyük észre, hogy a „szoftvertervezés” (mint műszaki, mérnöki munka) és a „projekttervezés” (mint menedzsment feladat) átlapolódnak az agilis környezetben!

Agilis szoftvertervező:

User experience a legfontosabb.

Felhasználó a középpontban:

UX (User Experience) design – a korábbi UI design.

Az UX elemei „agilisak”.

Agilis Business analyst:

User storykat ír és részleteket kidolgoz.

JUST IN TIME - fontos

Agilis fejlesztési technikák:

Extreme programming (XP):

A szoftverfejlesztés agilis módja, melyet bizonyos értékek, elvek és fejlesztési technikák írnak le.

Legfontosabb értékek a szoftverfejlesztés irányítására:

Kommunikáció, egyszerűség, visszajelzés, bátorág és tisztelet.

További útmutatók, alapelvek:

emberség, gazdaságosság, kölcsönös előnyök, hasonlóság, fejlődés, sokféleség, reflektió, flow, lehetőség, redundancia, minőség, kis lépések és vállalt felelősség.

XP tizenhárom elsődleges gyakorlatot ír le:

Üljünk együtt, teljes csapat, informatív munkakörnyezet, energikus munka, páros programozás, user story, heti ciklusok, negyedéves ciklusok, csúszás, tízperces build, continuous integration, „test first” programozás és inkrementális design.

Páros programozás:

Két programozó ugyanazon a munkaállomáson dolgozik.

Refaktorálás/ refaktORIZÁLÁS:

Egy meglévő forráskód belső struktúrájának fejlesztése / jobbá tétele úgy, hogy közben a program működése ne változzék.

TDD – Test Driven Development:

Tesztvezérelt fejlesztés, ciklikus, automatizált tesztek.

Általában XP-ben és agilis fejlesztésben használják.

- A programozók előbb a teszteket írják meg
- A tesztek kezdetben nem futnak le
- Rendre megírják a kódöt a tesztekhez
- A teszteket újabb elemekkel egészítik ki

Agilis fejlesztő

- Műszaki jellegű döntések
- Sok-sok tesztet ír
- Implementálja a user storykat
- Minőség centrikus gondolkodás
- Folytonos refaktorálás, fejlesztés
- A csapat egy tagja!



“Yes, you are a developer and yes, you’re agile but that doesn’t necessarily make you an agile developer.”

Tesztelés

Miért szükséges tesztelni?

A szoftverben hibák lehetnek, és vannak. Teszteléssel meg lehet találni őket.

Általában tesztelés után is a hibák 15%-a bent marad az ügyfélnek való átadáskor.

A gyakorlatban nem lehet minden 100%-osan tesztelni. A tesztelés célját, tartalmát, mértékét, időtartamát... minden az aktuális helyzet ismeretében kell meghatározni.

A tesztelés a szoftverminőség fontos eleme, de a jó minőség nem biztosítható csak teszteléssel.

A tesztelés definíciója

Egy fejlesztéssel létrehozott, szoftver rendszerrel kapcsolatos, tudatosan tervezett és végrehajtott tevékenység sorozat, ami a hibák minél előbbi megtalálására és kijavítására irányul. Továbbá a hibák okainak felderítésére és azok megelőzésére.

A TESZTELÉS NEM HIBAKERESÉS ÉS NEM TESZTEK FUTTATÁSA!

A TESZTELÉS A SZOFTVERFEJLESZTÉSI ÉLETCIKLUS MINDEN FÁZISÁBAN MEGJELENIK (követelmények fejlesztése, tervezés, kódolás, tesztelés, átadás)

Definíciók:

- **Programhiba:** A program olyan belső hibája, amely azt eredményezheti, hogy a szoftver nem tudja teljesíteni az elvárt viselkedését, azaz a program meghibásodásához vezethet. (bug, defect, fault)
- **Meghibásodás:** A komponens, illetve a rendszer eltér az elvárt eredménytől vagy szolgáltatástól. (failure)
- **Emberi eredetű hiba:** Emberi tevékenység, amely során helytelen eredmény jön létre. (error, mistake)
- **Hibakeresés:** A szoftver meghibásodás okainak megtalálási, analizálási és eltávolítási folyamata. (debugging)
- **Hibamaszkolás:** Olyan állapot, amikor az egyik hiba megakadályozza a másik hiba megtalálását. (defect masking, fault masking)
- **Előfeltétel:** Környezeti vagy állapotbeli feltételek, amelyeket teljesíteni kell, mielőtt egy komponensen vagy rendszeren tesztet vagy tesztelési folyamatokat kezdenénk. (precondition)

A szoftverfejlesztés 7 alapelve

1. Hibák látszólagos hiánya

Ha nem találunk hibát, az nem azt jelenti, hogy a rendszer tökéletes.

2. Nem lehetséges kimerítő teszt

Mindenre kiterjedő tesztelés nem lehetséges. Helyette kockázatelemzést és prioritásokat kell alkalmazni.

3. Korai tesztelés

A tesztelést a szoftver vagy rendszerfejlesztési életciklusban a lehető legkorábban el kell kezdeni, és előre meghatározott célokra kell összpontosítani.

4. Hibafürtök megjelenése

A tesztelést a feltételezett, illetve a megtalált hibák eloszlásának megfelelően kell koncentrálni.

5. A féregírtó paradoxon

Ha minden ugyanazokat a teszteket hajtjuk végre, akkor az azonos a tesztkészlet egy idő után nem fog új hibákat találni. A „féregírtó paradoxon” megjelenése ellen a teszteseteket rendszeresen felül kell vizsgálni, és új, eltérő teszteseteket kell írni.

6. A tesztelés függ a körülményektől (körülményfüggés)

A tesztelést különböző körülmények esetén különbözőképpen hajtják végre. Például egy olyan rendszert, ahol a biztonság kritikus szempont, más képp tesztelnek, mint egy e-kereskedelmi oldalt.

7. A hibamentes rendszer téveszméje

A hibák megtalálása és javítása hasztalan, ha a kifejlesztett rendszer használhatatlan, és nem felel meg a felhasználók igényeinek, elvárásainak. A követelményeknek való megfelelés és használatra való alkalmasság.

* Hibák csoportosulása: Kis számú modul tartalmazza a hibák legnagyobb részét.

A tesztelési folyamat elemei

Fő tevékenységek:

1. Tervezés és irányítás
2. Elemzés és műszaki tervezés
3. Megvalósítás és végrehajtás (végrehajtás fázisban: A tényleges és az elvárt eredmények összehasonlítása.)
4. A kilépési feltételek értékelése és jelentés
5. Teszt lezárása

Ezek BÁRMELYIK életciklus modellbe integrálódnak.

Tesztelés a CMMI-ben

3 folyamat: VER, VAL, PI. Mindegyik fejlesztési folyamat és ML3 szinten jelenik meg.

Fontos, hogy kódon kívül egyéb elemeket is tesztelni kell.

Verifikáció (VER): Célja annak biztosítása, hogy a munkatermékek teljesítsék a specifikált követelményeket.

CMMI-ben 3-as érettségi szinten.

Előkészítés verifikációra -> Egyenrangú szemlék végrehajtása -> Kiválasztott munkatermékek verifikációja

Validáció (VAL): Megmutatja, hogy a termék vagy termékkomponens a tőle elvárt módon fog-e működni, amikor a célkörnyezetbe kerül.

Előkészítés validációra -> Termék vagy termékkomponens validációja

Termék integráció (PI): Célja a termék összerakása a termékkomponensekből és annak biztosítása, hogy az integrált termék megfelelő módon működjön, valamint a késztermék átadása.

CMMI-ben 3-as érettségi szinten.

Termék integráció előkészítése -> Interfész kompatibilitás biztosítása -> Termékkomponensek összeépítése és a termék átadása

Tesztelés dokumentálása

A tesztelés minden elemét / tevékenységét dokumentálni kell.

Általában a következő dokumentumokat készítjük el a teszteléshez kapcsolódóan:

- **Tesztelési irányelvek**

Magas szintű dokumentum, amely a szervezet elveit, megközelítésmódját, valamint céljait mutatja be a tesztelésre vonatkozóan. (test policy)

- **Teszt terv (és ennek részletezése)**

A teszt hatáskörét, megközelítését, erőforrásait, valamint a tevékenységek tervezett ütemezését tartalmazó dokumentum. Ezen kívül meghatározza a tesztelemeket, a tesztelendő funkciókat, feladatakat, a tesztet végrehajtó személyek függetlenségét, a teszkörnyezetet, a műszaki tesztervezési technikákat, a belépési és kilépési feltételeket, valamint kockázatokat. (test plan)

A részletek:

Teszteljárás specifikáció / forgatókönyv / szcenárió: A teszt futtatásának tevékenységsorozatát rögzítő dokumentum. (test procedure specification)

Teszteset specifikáció: Egy tesztelekre vonatkozó, a teszteseteket leíró dokumentáció (cél, bemenetek, teszttévékenységek, elvárt eredmények, végrehajtás előfeltételei) (test case specification)

- **Teszt szcenárió / forgatókönyv**

- **Teszt eset**

Bemeneti értékek, végrehajtási előfeltételek, elvárt eredmények és végrehajtási utófeltételek halmaza, amelyeket egy konkrét céltól vagy a tesztért fejlesztettek. (test case)

Tesztesetek azonosítása: azonosító, cél, előfeltételek, input adatok, elvárt adatok, utófeltételek, teszt eset végrehajtásának története

- **Teszt szkript**

Legtöbbször teszteljárás specifikációra használt kifejezés, elsősorban automatizált teszt esetén. (test script)
Típusai: manuális, automatizált

- **Teszt adatok**

Olyan adat, amely a teszt előtt is létezik (például egy adatbázisban) és amely kölcsönhatásban van a tesztelés alatt álló rendszerrel, vagy a rendszerkomponenssel. (test data)

- **Tesztelési jegyzőkönyvek**

Tesztnapló: A tesztvégrehajtáshoz kapcsolódó részletek időrendi rögzítése. (test log, test record)

Összefoglaló tesztjelentés: A teszttévékenységeket és eredményeket tartalmazó dokumentum. Ebben a dokumentumban található a kilépési feltételeknek megfelelően ellenőrzött tesztelemek kiértékelése is. (test summary report)

Teszt-kiértékelési jelentés: A tesztfolyamat végén készített dokumentáció, amely összegzi az összes teszttévékenységet és eredményt. Ezen kívül tartalmazza a tesztfolyamat kiértékelését és a teszt során szerzett tapasztalatokat. (test evaluation report) -> E jelentés alapján születhet döntés a továbblépésről.

- **Hibalista, hibák leírása, hibajelentés**

Olyan dokumentum, amely leírja a szoftver azon hibáit, amelyek a program elégtelen működéséhez vezethetnek. (bug report, defect report)

- **Átadási –átvételi dokumentumok**

A végrehajtás során a következő információkat kell rögzíteni: A tesztelést végző személy nevét, a tesztelt funkciót, a tesztelés végrehajtásának dátumát, a tesztelés sikeres vagy sikertelen voltát, milyen gépen történt a tesztelés.

Hibák

Tesztelési hibák:

Ha a tesztelt szoftvernek a teszt során tapasztalt hibás viselkedését nem a tesztelt szoftver hibái okozzák.

Tesztelési hiba lehet:

- **Teszt specifikációs vagy tesztadat hiba:** A hibát a vizsgálati pont minősítésének hibás specifikációja vagy a rosszul megválasztott tesztadatok okozták.
- **A teszt hibás végrehajtása:** A hibát a teszt nem megfelelő végrehajtása (pl. a végrehajtás lépéseinél helytelen sorrendje, ütemezési hibák) okozták.
- **Hibás teszkörnyezet:** A hibát a teszkörnyezet rossz megválasztása (pl. hibás tesztprogram) okozta. Tesztelési hibák feltáráskor a hibát okozó elemet el kell hárítani (pl. javítani kell a teszt specifikációt, módosítani kell a teszt adatokat, a teszt környezetet, a teszt lépéseinél végrehajtási sorrendjét stb.), majd meg kell ismételni a tesztet.

Szoftver hibák:

A hibás viselkedést a tesztelt szoftver hibái okozzák.

Szoftver hiba lehet:

- **Megvalósítási hiba:** A szoftver nem teljesíti a rendszertervben előírtakat.
- **Tervezési hiba:** A hiba a nem megfelelő rendszerterv következménye.

Hiba felmerülése esetén a projekt vezetője dönt a további lépésekéről. Az ilyen döntéseket bizonylatolni kell (pl. bejegyzés a teszt naplóba, e-mail stb.)

Hibák dokumentálása:

Az egyes hibákról a következő információkat kell rögzíteni: hiba azonosító, hibajelenség leírása, tesztelt program azonosítója, teszeset azonosító, a hiba felfedezésének dátuma, a hibát felfedező személy, a hiba súlyossága, a hiba prioritása, a hiba állapota (rögzített, javítás alatt, kijavított), a hiba javításáért felelős személy, javítás dátuma.

Hibaelemzés:

A hiba elemzéshez általában négy fő paramétert használhatunk:

- A hiba aktuális állapotának státusza (nyitott, kijavított, lezárt, stb.)
- A hiba fontosságának prioritása.
- A hiba súlyossága.
- A forrás, ahol a hiba előfordult, és mi a hiba eredeti oka.

A hiba elemzéskor az alábbi kimutatásokat szokás elkészíteni:

- Hiba eloszlás riport: a hibák száma egy vagy két paraméter függvényében. (pl. a hibák száma prioritás vagy súlyosság szerint)
- Hiba korosítás riport: milyen sokáig marad javítatlanul egy hiba.
- Hiba tendencia riport: hibák száma státusz szerint az idő függvényében.
- Teszt eredmény és fejlődés riport: a tesztelés végrehajtásának eredményét mutatja meg az iterációk számán és teszt ciklusokon keresztül.

Teszt lezárása

A tesztlezárási fázisban gyűjtjük össze a tesztelés során előállított adatokat, hogy a tesztverből, számokból, tényekből és egyéb tapasztalatokból összegyűjtött adatokat konszolidáljuk. A tesztlezárási fázisban véglegesítjük és archiváljuk a tesztvert, értékeljük ki az eredményeket és készítjük el ő az összefoglaló tesztjelentést. (test closure)

OPEN ISSUES	PRIORITY	ASSIGNED	TEST CASE ID	Reported DATE	STATUS	Updated Date
1	High	QA Team	34	09-Apr-15	In Progress	13-Apr-05
2	Low	Dev Team	98	09-Apr-15	Resolved	13-Apr-05
3	Important	BA Team	198	09-Apr-15	Unresolved	09-Apr-05
4	Critical	Mangement	NA	09-Apr-15	Closed	10-Apr-15

CLOSED ISSUES	TEST CASE ID	CLOSURE DATE
1	87	13-Apr-05
2	45	13-Apr-05
3	NA	10-Apr-15

COMING Week Priorities	
Project ID:	W10978
Total test Cases to Cover:	109 to 450
Scheduled date:	14-May-15
	New requirements from Business
Reason for extending:	

Upcoming Task	
Project:	W10978
Scheduled Task:	New feature
Date of release:	29-May-15

BUGS/DEFECT SUMMARY	
ACTIVE DEFECTS	40
CLOSED DEFECTS	8
EXECUTED TEST CASE	298
UN EXECUTED TEST CASE	498

Tesztek típusai

Teszt típusok egymásra épülése:

- Teszteljük az egységeket / komponenseket.
- Integráljuk a komponenseket alrendszerbe, teszteljük ezeket.
- Folytassuk az integrációt, míg nem a teljes rendszert tesztelni tudjuk.

Teszt típusok a tesztelt elem szerint:

- Egyégeszt / komponens teszt / unit test:

Komponens: a legkisebb önállóan tesztelhető szoftver egység

Az egységeket / komponenseket teszteljük a specifikációval szemben. Egy egység: egy v. több eljárás, függvény.

A forráskód egy adott egységének helyességét vizsgáló teszt-módszer.

Alapötlet: készüljön teszt eset a modul minden nem triviális függvényére, funkciójára vagy metódusára, úgy, hogy minden teszt eset önálló, a többitől különválasztható legyen.

- Integrációs teszt:

Az egységeket integráljuk, és az integrált részek interfészeit teszteljük a specifikációval szemben.

Általában az egységesztet követi és a rendszertesztet előzi meg Az egységeszten sikeresen „átesett” modulokat veszi alapul.

Célja, hogy a nagyobb elemekkel szemben támasztott, funkcionalitásra, teljesítményre, megbízhatóságra vonatkozó követelményeket ellenőrizze.

A modulok közötti interfészek kerülnek kipróbálásra. A modulok / alrendszerök közötti együttműködést szimulálva történik a tesztelés.

Alapötlet: minden tesztelt, ellenőrzött alapelemekből építkezünk.

Nagy bumm teszt: az integrációs tesztelés egyik fajtája, ahol a szoftver és a hardver elemeket akár egyetlen rendszerbe integrálva teszteljük (big-bang testing)

- Rendszerteszt:

A teljes rendszert teszteljük a specifikációval szemben.

Input a teszthez: integrált, sikeresen tesztelt (al)rendszer(ek).

A rendszertesztet a felhasználónak is értenie kell – gyakran szorosan kapcsolódik az átvételi teszthez.

Teszt típusok a tesztelésben részt vevők szerint:

- **Tervezői teszt**
- **Fejlesztői teszt**
- **Felhasználói teszt**
- **Közösen, több érdekkelt fél által végzett teszt (közös szemlék)**

Teszt típusok a teszt eredményének felhasználása szerint:

A tesztelés eredményeképpen valamelyen (hivatalos) döntés születik / lépés történik.

- **Átvételi teszt:** a felhasználó, vagy a megrendelő által a végterméken végzett feketedoboz teszt, amely azt hivatott eldönten, hogy megfelel-e a termék a megfogalmazott (üzleti) elvárásoknak, illetve folyamatoknak. (acceptance testing)
- **Üzemelteszt:** A szoftvertermék minden kiadásakor az üzemeltetőnek üzemi (operational) tesztet kell végrehajtania és ha az előírt kritériumok teljesülnek, a szoftverterméket üzemi használatra ki kell adni. Az üzemeltetőnek biztosítani kell, hogy a szoftverkód és az adatbázis a tervben leírtak szerint induljon el, működjön és álljon le.
- **Installációs tesztelés:** A fejlesztői környezeten kívül végzett tesztelés.

Teszt típusok egyéb kritériumok szerint:

- „Kézi” tesztelés
- **Automatizált tesztelés:** Felhasználó által készített szkriptek alapján történő tesztelés. Tesztelési eszközök használó tesztelés.
- **Terheléses teszt** (Load testing): Általában azt jelenti, hogy a szoftver várható alkalmazását úgy modellezik, hogy az egy időben sok felhasználó jelenlétét szimulálja. Az ilyen tesztelés a többfelhasználós rendszerekben hasznos. Ha a terhelés a normálisan elvárt határ fölött emelkedik, akkor stressz tesztről beszélünk.
- **Teljesítmény teszt** (Performance test): Annak megállapítására, hogy a rendszer bizonyos aspektusai milyen gyorsan „nyilvánulnak meg” egy bizonyos terheléskor.

Tesztelési szintek:

A tesztelési szintek a szoftverfejlesztési életciklus szerinti csoportosításai a teszteknek. Az ISTQB a következő 4 szintet említi:

- Komponens (Unit) teszt
- Integrációs teszt
- Rendszer teszt
- Átvételi teszt

Tesztelési szintek



Teszt típusok az alkalmazott technika szerint:

Általában a szoftverfejlesztési életciklusban először a statikus tesztelés, majd a strukturális tesztelés, végül a funkcionális tesztelés technikái jutnak szerephez.

Különböző szinteken különböző technikákat használunk. Bármely szint esetében igaz, hogy az levárt eredményeket többféle technika ötvözésével lehet elérni.

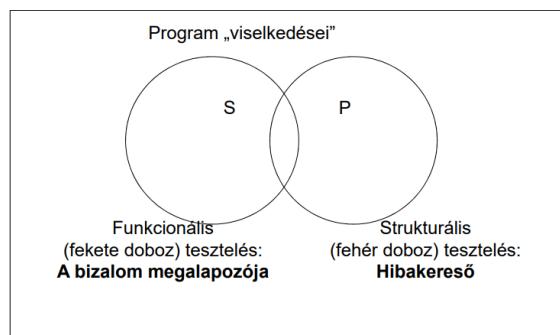
Tesztelési technikák: Segítenek a megfelelő tesztelési módot / eseteket kiválasztani.

Alapvető megközelítések:

- A szoftver belső szerkezetének / felépítésének vizsgálata (a program futtatása nélkül): statikus tesztelés
- A szoftver működés közbeni vizsgálata: dinamikus tesztelés

Általában a statikus és dinamikus, az utóbbin belül pedig a strukturális és funkcionális technikákat megfelelő arányban kombinálva alakul ki az adott esetben jó tesztelési módszer.

Teszt esetek azonosítása:



• Statikus tesztelés:

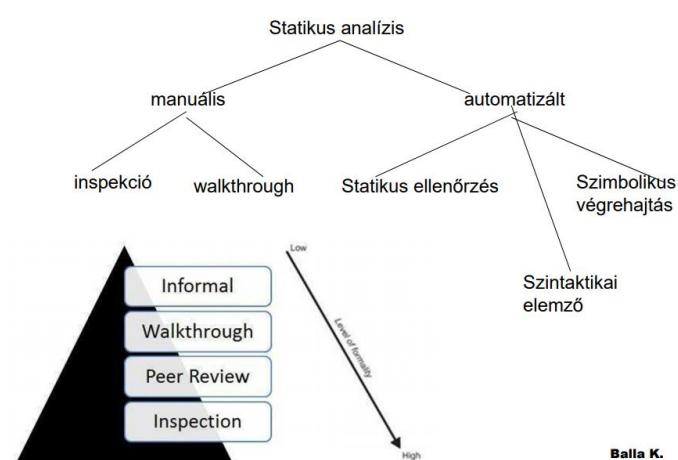
A tesztelésnek az a formája, amikor magát a szoftvert nem használjuk (nem futtatjuk).

Statikus tesztelési módszerek: kódszemlézés, inspekció, „walkthrough”, statikus kódelemzés.

Mindegyik emberi „megfigyelésen” alapul. Egy komponens vagy rendszer tesztelése specifikáció, vagy implementáció szinten a szoftver futtatása nélkül.

Alapelt: futtatással nem tudunk minden lefedni – viszont a teljes kódot át lehet nézni.

Cél nemcsak a hibák megtalálása, hanem okaik felderítése és további előfordulásuk meggyójtása (pl. a folyamat javításával)



Statikus kódelemzés: A szoftver forráskódjának elemzése azzal a céllal, hogy megértsük, mit csinál a szoftver. Ugyanakkor, helyességi kritériumokat is felállítunk.

Statikus analízis: Formális módszer-család, amellyel egy szoftver viselkedéséről automatikusan információ nyerhető. Pl: debugger

Statikus analízis technikák: a program belső szerkezetének vizsgálata / a program teljességének és konziszenciájának vizsgálata előre meghatározott szabályok alapján / a program összehasonlítása a specifikációjával vagy dokumentációjával.

A statikus analízis eredménye nem egyértelmű: tulajdonképpen semmilyen módszerrel nem lehet eldöntheti, hogy futtatás során megjelennek-e hibák.

Példák:

- Ellenőrzés papíron: A szoftver, vagy a specifikáció tesztelése a végrehajtás kézi szimulálása által.
- Walkthrough/átvizsgálás: Egy dokumentum szerzője által végzett lépésenkénti bemutató abból a célból, hogy információt gyűjtsön, valamint közös álláspontot alakítson ki.
- Inspekción/felülvizsgálat:
Az inspekció a leghatásosabb az összes szemle között – viszont költséges és nehéz bevezetni.
Elmaradhat az egységeszt és integrált teszt, csak a rendszertesztet kell elvégezni.
A Fagan-féle inspekció lényeges eleme az inspekcióban részt vevő szerepkörök definiálása.
(moderátor, olvasó, szerző, jegyző, inspektorok)
Peer review / egyenrangú szemle / egyenrangú felülvizsgálat: A szoftverfejlesztés alatt végzett tevékenységek felülvizsgálata nem a tevékenységet elvégző által, melynek célja, hogy hibákat fedezzen fel, illetve javító javaslatokat hozzon.

- **Dinamikus tesztelés:**

A rendszer futtatásának próbája, teszkörnyezetben, tesztadatokkal.

Típusai:

- **Feketedoboz teszt**: Követelményeken alapuló, funkcionális, vagy nem-funkcionális teszt.
Feketedoboz tesztervezési technika: olyan módszer, amelynél a szoftver specifikáció alapján, a program belső szerkezetének ismerete nélkül tervezünk teszteket. (pl: határérték tesztelés, döntési táblák, állapotátmenet tesztelés...)

A funkcionális tesztelés az input adatokból kiindulva vizsgálja az outputot.

Határérték elemzés: A legkezdetlegesebb tesztelési forma. Független változókat és fizikai mennyiségeket feltételez. „Egyszerre egy hiba” elve. A robosztus teszttel, valamint a „legrosszabb eset” teszttel kombinálva jó eredményeket ad. A robosztussági teszt jól használható a belső változók tesztelésére. Jól használható a hibaüzenetek figyelésére.

Általánosításai: Változók számának növelése. A korlátok kiterjesztése / általánosítása.

Ekvivalencia particionálás: Ekvivalencia osztályok létrehozása, a teszt esetek számának csökkentésére: gyenge normál, gyenge robosztus, erős normál, erős robosztus.

Az ekvivalencia osztály alapú tesztelés segítségével szeretnénk biztosítani, hogy a tesztelésünk „teljes”, és segítségével szeretnénk elkerülni a redundáns adatokkal való tesztelést.

Jellemzően minden egyes ekvivalencia partíciót érdemes legalább egyszer lefedni.

Kulcsfontosságú: Az ekvivalencia osztályokat (az ekvivalencia relációt) jól, okosan kell meghatározni!

Döntési táblákon alapuló tesztelés: A döntési táblák jól használhatók olyan esetek leírására, amikor különböző akciók kombinációira kerül sor bizonyos, változó feltételrendszer mellett. (a változók közötti logikai összefüggésekkel is figyelembe veszik)

Példa

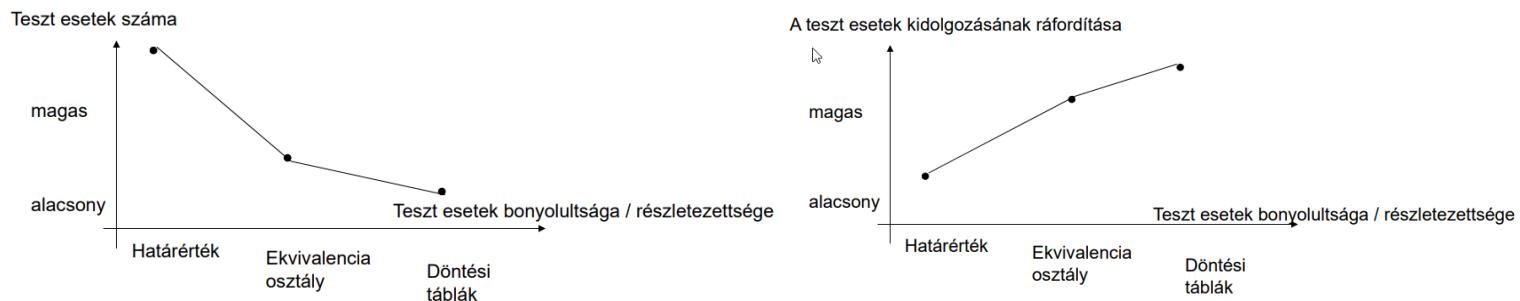
	Szabály1	Szabály2	Szabály3	Szabály4	Szabály5	Szabály6
c1(cond)	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	-	T	F	-
A1(action)	X	X		X		
A2	X				X	
A3		X		X		
A4				X		X

C: feltétel

A: akció / tevékenység

Állapotátmenet teszt: Úgy tervezük meg a teszteseteket, hogy érvényes és érvénytelen állapotátmeneteket generáljanak.

Használati eset teszt: A műszaki tesztereket (test design) különböző használati eset forgatókönyvek futtatására készítették. (use case testing)



- **Strukturális/fehérdoboz teszt:** A szoftver belső szerkezetén/forráskódon alapul. A belső struktúrára vonatkozó információk a kódból, architektúrából, modellekben nyerhetők. A strukturális (fehérdoboz) teszt minden tesztszinten végrehajtható. A lefedettség azt mutatja meg, hogy egy teszkészlet minden mértékben hívott meg egy struktúrát, és ezt az értéket a lefedett elemek százalékában fejezik ki.

- **Utasítás szintű teszt és lefedettség:** Annak értékelése, hogy valamely teszeset készlet a futtatható utasítások hány százalékát érintette.
- **Döntési teszt és lefedettség:** Annak értékelése, hogy valamely teszeset készlet a döntési eredmények (pl. egy If utasítás Igaz vagy Hamis lehetőségei) hány százalékát hajtotta végre. A döntési lefedettséget a megtervezett, illetve végrehajtott döntési eredmények száma és a tesztelendő kódban található összes lehetséges döntési eredmény hányadosa határozza meg.
- A döntési lefedettség magasabb rendű az utasításlefedettségnél: 100%-os döntési lefedettség esetén garantált a 100%-os utasítás-lefedettség, aminek fordítottja nem igaz.

Technikák:

- **Útvonal alapú tesztelés:** A teszeseteket úgy tervezük, hogy egy-egy végrehajtási utat járjanak be. A programnak megfeleltetünk egy programgráfot. (path testing)
- **Változtatáshoz kapcsolódó tesztelés:** Bármikor, ha változtatunk a rendszerben. Regressziós tesztelés: Egy korábban már letesztelt program, módosítást követő tesztelése, annak biztosítása érdekében, hogy a módosulás nem okozott hibát a szoftver nem módosított részeiben.
- **Progressziós tesztelés:** feltételezzük, hogy az integrációs teszt rendben lefutott, és az új funkciókat tesztelhetjük. A progressziós tesztek kifejezetten az utolsó változtatásnál módosított/létrehozott részek/funkciók tesztelésére koncentrálnak
- **Karbantartási teszt:** Módosítások vagy megváltozott környezet miatt a működő rendszeren végrehajtott teszt. (maintenance testing) Módosítások lehetnek: tervezett kiegészítő változtatások, javító vagy vészhelyzeti változtatások.

- **Egyéb tesztelési technikák:**

- **Sajátos értékek tesztelése (Ad-hoc tesztelés):** A tesztelő saját szaktudását, korábbi tapasztalatát felhasználva tapint rá kényes pontokra.
- **Random tesztelés:** Ötlet: véletlenszám-generátort használva válasszunk input értékeket.
- **Tapasztalat-alapú teszt:**
Hibasejtés: olyan műszaki tesztervezési módszer, amely során a tesztelők tapasztalata alapján próbálják megsejteni a tesztelendő szoftverben levő hibákat, illetve ez alapján próbálnak megfelelő teszteket tervezni.
Hibatámadás: célzott próbálkozás a teszt tárgyának minőségének, különösképpen a megbízhatóságának meghatározására azáltal, hogy speciális meghibásodásokat próbálunk meg szándékosan előidézni.
Felderítő teszt: informális tesztervezési módszer, amely során a tesztelő aktívan felügyeli a tesztek tervezését, a futtatás során szerzett információkat összegyűjt és hasznosítja új és jobb tesztek tervezése érdekében. (exploratory testing)
Ellenőrző lista alapú teszt: ami során a tapasztalt tesztelő magas szintű tesztelemeket jegyez fel, ellenőriz, vagy szabályokat és kritériumokat használ a program verifikáláshoz. (checklist-based testing)

Tesztmenedzsment

A teszttevékenységek tervezése, becslése, monitorozása és irányítása, amelyet általában a tesztmenedzser végez.

Például: a tesztelés hatékonyságának figyelésére

A tesztelés hatékonysága

Hibamegtalálási százalék (DDP):

Ebben a tesztelésben megtalált hibák száma	X 100	„Ez a tesztelés” lehet:
Az összes hibák száma, beleértve azokat, amelyeket csak később találunk meg		<ul style="list-style-type: none"> - egy teszt-fázis (pl. egységeszt, integrációs teszt, átvételi teszt....) - egy funkcióra vagy alrendszerre vonatkozó összes teszt - egy rendszerre vonatkozó összes teszt

A még a rendszerben levő hibák számának előrejelzése.

Tesztelés agilis környezetben

Folyamatosan vannak rövid iterációk a tervezés, kódolás és tesztelés tevékenységekre.

A tesztelési tevékenységek is iteratív módon, folyamatosan kerülnek végrehajtásra.

Agilis tesztelés: Az agilis tesztelési gyakorlatok az agilis projektvégrehajtáshoz igazodnak: extreme programming (XP), a fejlesztést a tesztelés vevőjének tekintik, hangsúlyozzák a „test-first” tervezési paradigmát.

Az agilis tesztelés a lehető legkorábbi tesztelést hangsúlyozza a szoftverfejlesztési életciklusban.

Megköveteli a nagyon hangsúlyos vevői részvételt a tesztelésben is, amint a kód hozzáférhetővé válik.

A kódnak elég stabilnak kell lennie ahhoz, hogy lehetővé tegye a rendszertesztelest.

Fontos a regressziós tesztelés.

A kommunikáció a csapaton belül és a csapatok között kulcsfontosságú!

Tesztvezérelt fejlesztés: Test Driven Development/TDD

Általában az extrém programozásban és az agilis fejlesztésben használják.

A programozók először a teszteket írják meg. A tesztek kezdetben sikertelenek lesznek.

Rendre megírják a tesztekhez a kódot. A teszteket kiegészítik.



Folyamatos integráció: Continuous integration

Egy szoftver fejlesztési módszer, melyben a fejlesztőcsapat tagjai az általuk írt kódot legalább napi rendszerességgel integrálják a korábbi fejlesztések közé, ez napi többszöri integrálást jelent.

Minden új kód integrálása során automatizált tesztek ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kód részlet és ennek eredményeként a lehető leghamarabb visszajelzést ad az integráció eredményéről.

Automatizált build és tesztelés naponta történik; az integrációs hibák korán és gyorsan kiderülnek.

Az agilis tesztelők automatizált teszteket futtatnak, és gyors visszajelzést adnak a csapatnak a kód minőségéről.

Az eredményeket minden csapattag látja.

Az agilis tesztelő:

A tesztelőket bevonják a tervezésbe. Korai fázistól kezdve dolgoznak, a projekt teljes idején.

Átvételi / elfogadási kritériumok:

Meghatározás – Segít az ügyfélnek

Automatizálás – Implementálja

Tesztcsapat: Tevékenységének célja a lehető legtöbb külső hibát azonosítani, és az ezeket okozó belső hibákat azonosítani és kijavítani.

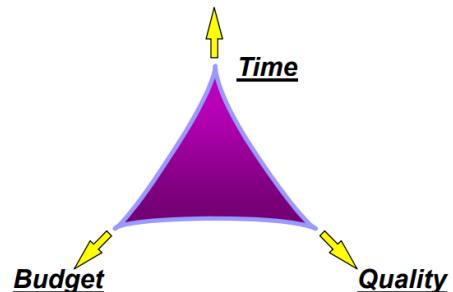
A szoftverprojektek menedzsmentje

Miért szükséges a projektek menedzselése a szoftverfejlesztésben?

- Mert a szoftvert nem egyénileg, hanem csapatban fejlesztik.
- Mert a szoftverfejlesztés projektben történik.
- Nincs egyetlen személy, aki az adott szoftverfejlesztés minden részletét ismerné, mégis, valakinek koordinálnia kell a csapatok / csapattagok együttműködését.
- A menedzsmentre azért van szükség, hogy a szoftverfejlesztési projektben folyó munkát koordinálja.

Projektmenedzsment (PM)

- Képességek, eszközök és technikák alkalmazása projekttevékenységekre azzal a céllal, hogy a projekt követelményeit teljesítsék.
- 20. század közepétől önálló szakma.
- Projektmenedzsment folyamatok a következő csoportokba tartoznak:
 - Előkészítés (Initiating)
 - Tervezés (Planning)
 - Végrehajtás (Execution)
 - Követés és vezérlés (Monitoring and Controlling)
 - Lezárás (Closure)
- PM tudásterületek: Integráció, Hatókör, Idő, Költség, Minőség, Beszerzés, Humán erőforrások, Kommunikáció, Kockázatmenedzsment, Érintett felek menedzsmentje
- **Projekt:**
 - o Egymáshoz kapcsolódó tevékenységek és erőforrások menedzselt halmaza, mely embereket is magába foglal, és egy vagy több terméket vagy szolgáltatást ad át egy megrendelőnek vagy végfelhasználónak. Van kezdete és vége. A projektek tipikusan terv szerint haladnak.
 - o A projekt egy időben, költségekben és erőforrásokban korlátozott, adott követelményeknek megfelelő cél érdekében kezdő és végidőpontokkal ellátott koordinált és kontrollált tevékenységek halmaza.
 - o A projekt a kitűzött CÉL érdekében kölcsönösen egymásra ható tevékenységek csoportja, amely magában foglal idő-, költség-, erőforrás- és minőségtényezőket.
- Mi szükséges a sikeres projektirányításhoz?
Sok, összetett és egymással is kölcsönhatásban levő tényező ismerete, megértése.
Rátermettség, fokozott figyelem.
Útmutató, hogy a sok tényező közül mindegyikkel, a megfelelő időben foglalkozzunk, vagyis egy jó MÓDSZERTAN.

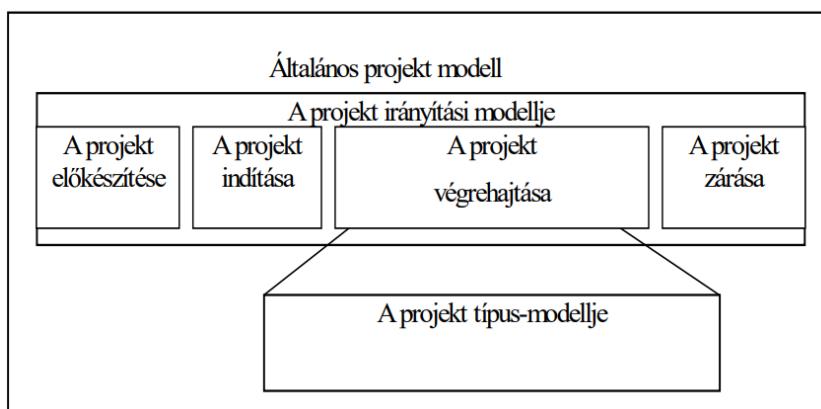


Projektirányítási módszertanok

- Útmutatást nyújtanak a projektszervezetre és – menedzsmentre vonatkozóan.
- Termékszemléletűek: minden tevékenység valamilyen terméket állít elő / módosít.
- Az előállított termékeknek határozott minőségi követelményeket kell kielégíteniük, ezért a minőségbiztosítás a módszertanok keretében kifejezett hangsúlyt kap.
- Szemléletükben a projektnek véges élettartama, megadott felelősségi körökkel rendelkező szervezeti struktúrája, meghatározott és egyedi termékei, a termékek előállításához szükséges tevékenységei, a tevékenységek elvégzésére alkalmas erőforrásai van(nak).
- A projekt szakaszokra bomlik. A szakaszok vezetői szempontból különálló egységet alkotnak. A szakasz végét a benne meghatározott termék előállítása jelenti, ha kielégíti a megállapodás szerinti minőségi feltét
- A PMBOK szerint a projektmenedzsment alapvetően az alábbi 9 terüettel foglalkozik:

Integráció-menedzsment	Terjedelem-menedzsment	Ütemezés-menedzsment
Költség-menedzsment	Minőség-menedzsment	Emberi erőforrás-menedzsment
Kommunikáció-menedzsment	Kockázat-menedzsment	Beszerzés-menedzsment

- Közös elemek valamennyi PM módszertanban:
 - o Modellezik a projektet.
 - o Struktúrát adnak a menedzsment tevékenységekre. Alapvetően: tervezés, követés, vezérlés.
 - o Útmutatást adnak arra vonatkozóan, hogy ezeket tevékenységeket hogyan kell végrehajtani.



- Alapvető PM tevékenységek:
 - Mindig: Plan-Do-Check-Act
 - Projektindítás
 - Becslés
 - Tervezés
 - Követés és vezérlés (~ ellenőrzés és cselekvés)

CMMI-ben

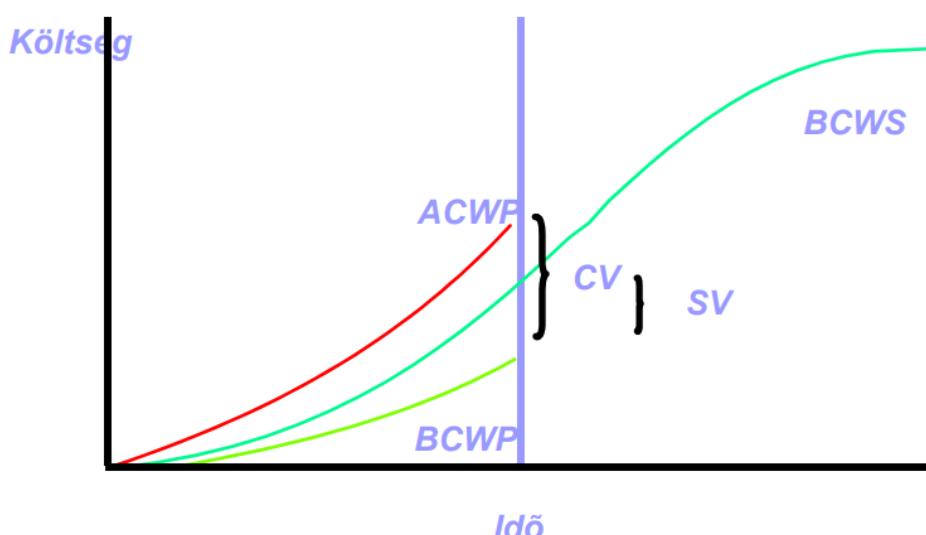
- Projekttervezés (PPL):
 1. Becslések végzése
 2. Projektterv kialakítása
 3. A tervhez való elkötelezettség kialakítása
- Projektkövetés és –vezérlés (PMC):
 1. Projektkövetés a terv alapján
 2. Helyesbítő intézkedések menedzselése

A PPL a
CMMI-ben 2-
es érettségi
szintű
folyamat.
Alap!

A PMC a
CMMI-ben 2-
es érettségi
szintű
folyamat.
Alap!

PMC: A teljesítménymérés fogalmai:

- **ACWP** - Felhasznált költségek (Actual Cost of Work Performed)
- **BCWS** - Adott időszakra ütemezett munkára tervezett költség (Budget Cost of Work Shceduled)
- **BCWP** - Megvalósult érték = Elvégzett munkára tervezett költség
- **CV** - Elvégzett munka alapján számolt költségeltérés (Cost Variance; $CV = BCWP - ACWP$)
- **SV** - Elvégzett munka és eltelt idő alapján számolt költségeltérés (Schedule Variance; $SV = BCWP - BCWS$)
- **CPI** - Költség szerinti teljesítmény index (Cost Performance Index; $CPI = BCWP / ACWP$)
- **SPI** - Időzítés szerinti teljesítmény index (Schedule Performace Index; $SPI = BCWP / BCWS$)



PMC: A teljes befejezéskori költség becslése a követés során:

- A viszonyítási alap szerinti befejezésen alapuló becslés (Ezután terv szerint folytatódik javuló teljesítménnyel ...)
- A jelenlegi teljesítmény folytatásán alapuló becslés (Az idő tényezővel nem foglalkozunk...)
- A jelenlegi CV és SV kombinált előrevetítésén alapuló becslés (Az idő is pénz...)

A számolási képlet: ACWP + (tervezett költség -BCWP) / (CPI*SPI)

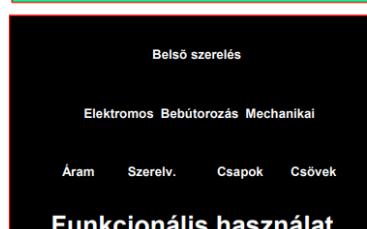
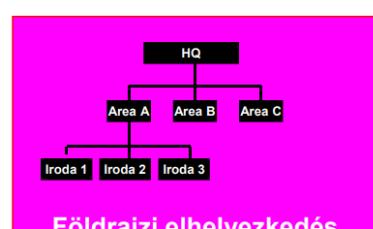
Az előbbi gyors becslési technikákat nem szabad végső becslésre használni csak a más módszerekkel kapott eredmények ésszerű ellenőrzésére.

PM és szoftverfejlesztés

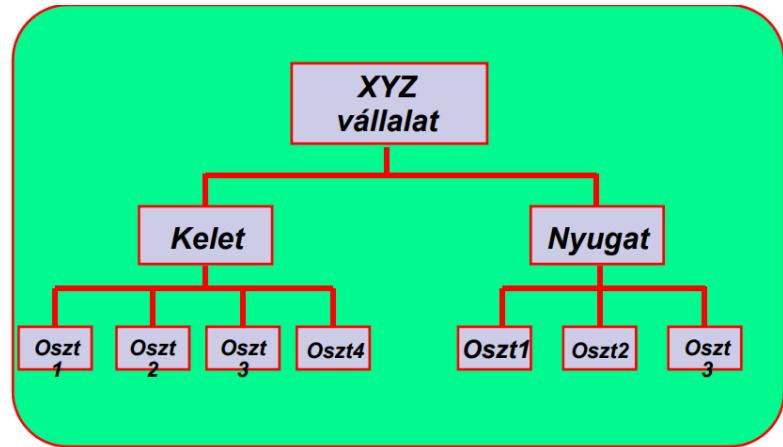
- Projekt előkészítése: Megvalósíthatósági vizsgálat, a projekt megértése, ajánlatkészítés.
- Becslés:
 - Becsülni: A projekt hatókörét (scope), időt, ütemezést, szükséges erőforrásokat, ráfordítást és költséget, a készülő szoftver méretét.
 - Becslések típusai: Nagyságrendi, félleg részletezett, részletes.
 - Csak korábbi / historikus adatok alapján végezhető!
 - A PSP szerint a cél, hogy konzisztensek legyünk!
 - **Költségbecsülés:**
Alapja: A végrehajtandó feladatok száma és bonyolultsága / Hasonló feladatokkal kapcsolatos tapasztalat / Rendelkezésre álló erőforrások száma / Rendelkezésre álló erőforrások jellemzői
Historikus adatok segítenek!!!
Mindig jegyezzük fel a becslés alapját!!!
 - **Tartalék:** A tartalékképzés felkészülés a becslés eredményétől való olyan eltérésekre, amelyek valószínűleg előfordulnak, de amelyeket nem lehet pontosan azonosítani a becslés készítésének időpontjában.
- Szerződés: Törvény által érvényesíthető igéretek. Típusai: Fixáras, költségtérítéses, a kettő kombinációja.
- A projekt indítása: Projekt indító gyűlés („kick-off” meeting”), PID: napirend, résztvevők.
- A projekt tervezése:
 - **Projekt terv:** A projekt tevékenységek végrehajtásának és ellenőrzésüknek az alapja; minden tevékenység célja a projekt vevőjének követelményeit kielégíteni.
 - A projekt tervezés magába foglalja a munkatermékek és folyamatok paramétereinek becslését, a szükséges erőforrásokat, az egyeztetéseket, az ütemterv elkészítését, a projekt kockázatainak felismerését és elemzését.
 - A projekt terve: Vezetői dokumentum, amely elmondja az elveket, a taktikát, az eljárásokat és a célkitűzéseket. Magában foglal hálóterveket, diagrammokat és lebontási szerkezeteket. A tervben szereplő hiba vagy mulasztás a projekt meghiúsulásához vezethet.
 - Időzítés: Logikai háló, sávdiagram
 - Erőforrások: Projekt erőforrás terv, részleg erőforrás terv
 - Költségvetés
 - Mérföldkő terv, MS meghatározási táblázat
 - Felelősségek hozzárendelése
 - Feladatlebontási struktúra: A WBS a projekt hierarchikus felbontása természetes elemi egységekre vezetési és követési célból.

Lehetséges közelítések a WBS

létrehozásához:

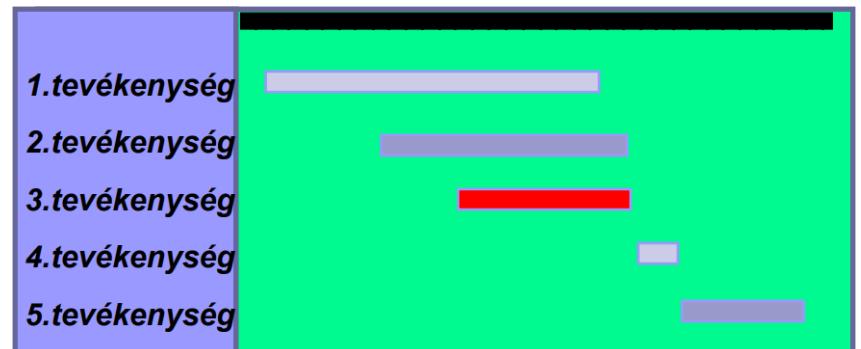


- Szerveztelebontási struktúra:

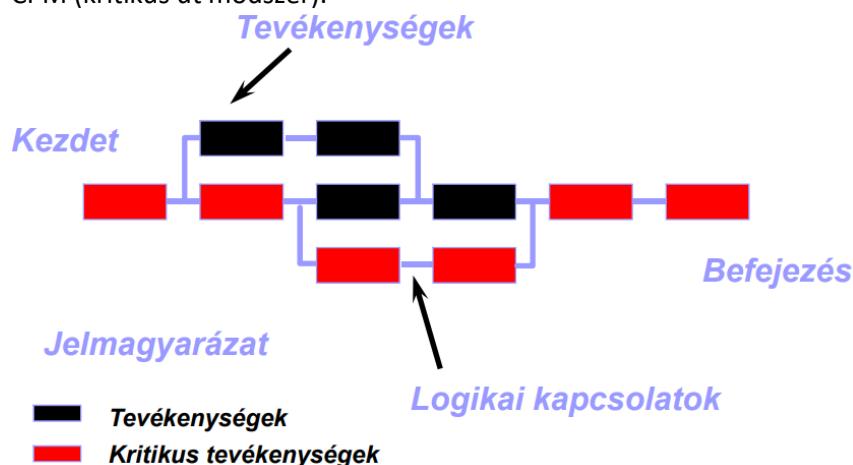


- Tervezési és ábrázolási technikák:

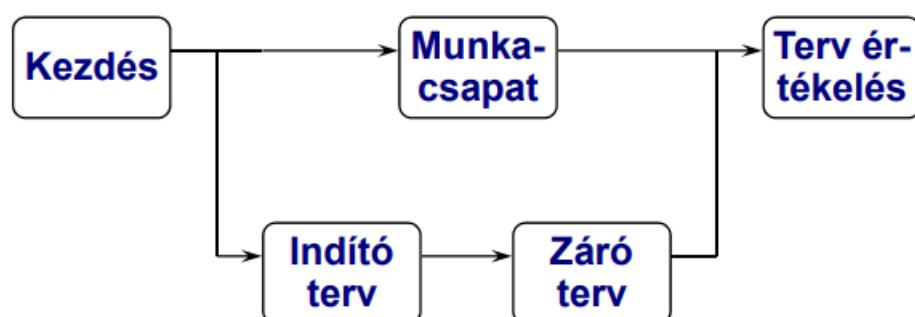
- Gantt (sávdiagram)



- CPM (kritikus út módszer):



- PERT (program kiértékelést szemléltető technika)
- Elsőbbségi háló: „tevékenység a csomópontron“ diagram, amelyen egy-egy tevékenységet egy doboz jelöl, a dobozok közötti nyílak a kényszert jelentő logikai összefüggéseket mutatják.

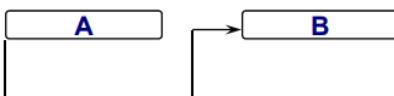


- Logikai megkötések / összefüggések:



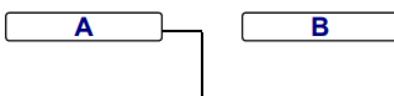
BEFEJEZES - KEZDET (FS)

B csak akkor kezdődhet el,
ha A befejeződött



KEZDET- KEZDET (SS)

B csak akkor kezdődhet el,
ha A elkezdődött



BEFEJEZÉS - BEFEJEZÉS (FF)

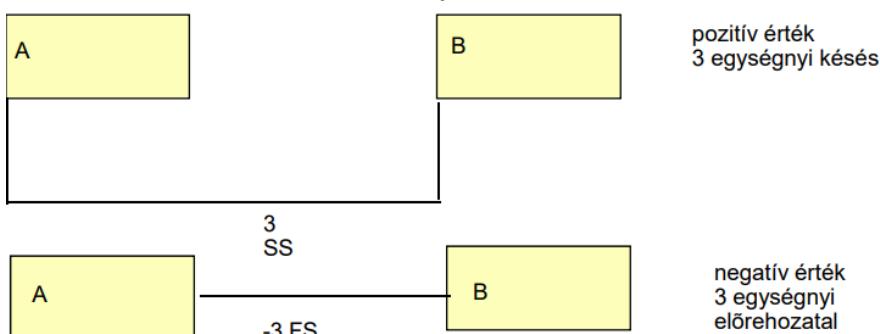
B csak akkor fejeződhet be,
ha A befejeződött



KEZDET- BEFEJEZÉS (SF)

B csak akkor fejeződhet be ,
ha A elkezdődött

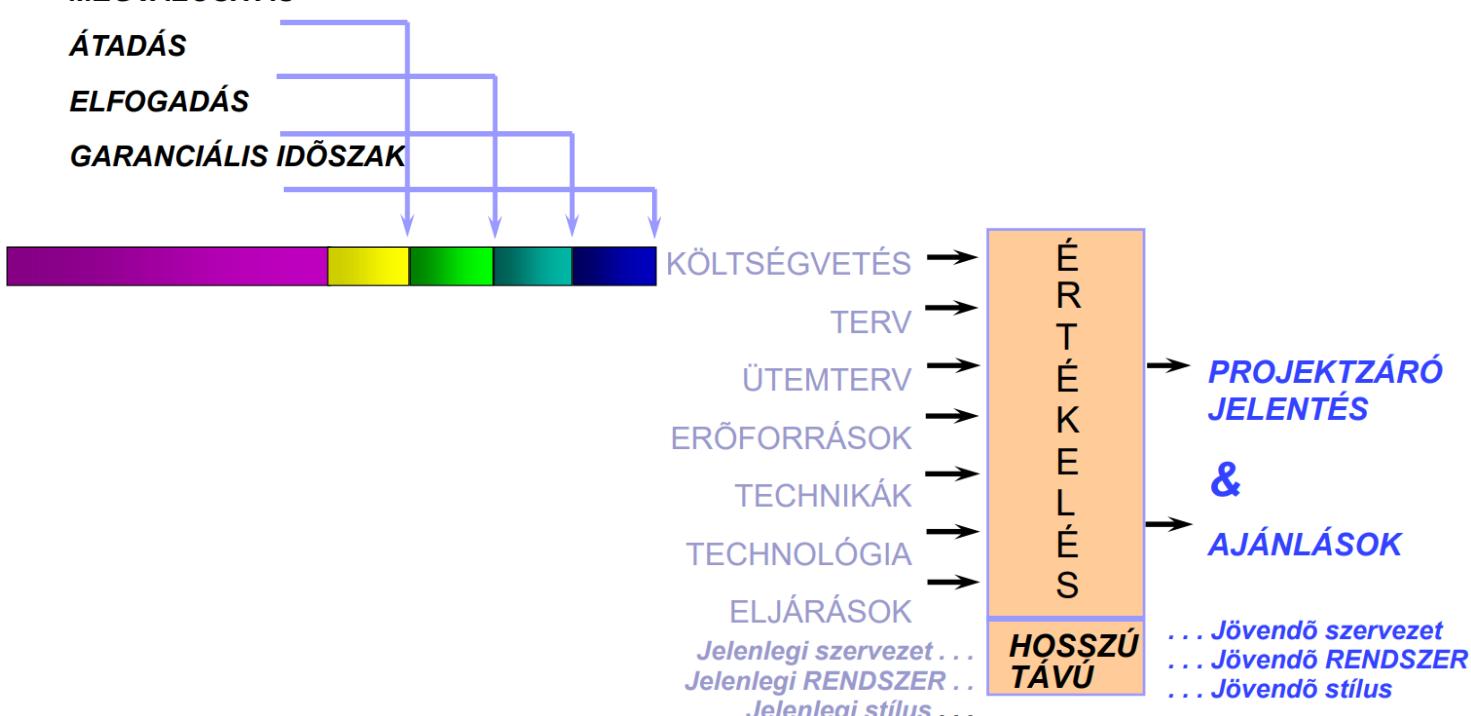
- Késleltetések / időbeli előrehozások jelölése:



- Időelemzés – Időterv: A tevékenységek időtartama és az első tevékenység kezdő dátuma alapján kell a legkorábbi és legkésőbbi kezdési, illetve befejezési időpontokat kiszámítani. Két irányban haladva: előre és visszafele.
- Erőforrás tervezése: Egyensúly kialakítása a szükséges és a hozzáférhető erőforrások között.
- Költségháború: Erőforrások költsége, tevékenységhez rendelt fix költség, mikor fog a költség felmerülni? ...

- **Projektzárás:**

MEGVALÓSÍTÁS



Projektmenedzsment agilis környezetben

Ugyanazokat a PM feladatokat kell végrehajtani, mint egy hagyományos környezetben (Agilis PM technikák használatával).

A Scrum-ot „az” agilis PM módszertanként emlegetjük.

Agilis tervezés: A projekttervezési és -követési tevékenységeket agilis projektekben gyakran a műszaki tevékenységekkel közösen végzik. Agilis projektekben nincs éles határ a menedzsment és a technikai feladatok között!

Agilis csapat: mindenki minden feladatban részt vesz / tájékozott róla.

Product Owner: Egy személy. Meghatározza a termék funkciót (=Product Backlog). Lehet befolyásolni.

Scrum Master: Egy személy, de nem projekt menedzser, hanem inkább coach. A folyamatért felelős, biztosítja, hogy az követve van. Biztosítja, hogy a team produktív és funkcionális.

Scrum Team: Kereszfunkcionális, 7 plusz/mínusz 2 fős csapat. Szakértők. Felelős a becslésért és a munka elvégzéséért, kiválasztja a sprint célját és a munka eredményét.

Csirkék és disznók: A Csirkék gyakorlatilag a menedzsmentet jelölik. minden olyan ember, aki nem a csapat része, de valamilyen értelemben kölcsönhatásban van a projekttel. A Disznók – maga a csapat.

Agilis becslés: Wideband Delphi, Planning Poker

Story Point vagy komplexitás pont: egység nélküli, a komplexitás és a méretet relatív nagyságát mondja meg.

Product Backlog / Termék teendőlistája: Funkciók, feladatok listája. Priorizált, becsült story-k. A magasabb prioritású elemek részletesebben kifejtettek. Az összes csapat egy listát használ. Lehetnek rajta: Story, Bug, Dokumentáció, Prototípus gyártás...

Sprint Backlog / Futam teendőlistája: A csapat által kiválasztott Product Backlog elemek egy Sprintre. A csapat és a PO megegyezik a tartalmában. Sprint alatt nem változik, be van fagyaszva a Sprint hosszára. Az elemek le vannak bontva Taskokra.

DoD (Definition of Done): Egy mindenki által elfogadott lista, amely tartalmazza, hogy mikor kerül egy user story „elvégzett” (Done) állapotba a sprint végén.

Burndown chart: A sprintben elkészült/visszamaradó feladatokat mutatja. Ez az alapértelmezett státusz mutató eszköz.

Burnup chart: Előrehaladás mérője. A legegyszerűbb esetben, két vonal: A teljes munka vonala („projekt scope” vonal), Az elégzett munka vonala.

Megbeszélések / Meetings: Story gyűjtő meetingek – milyen funkciók következnek, milyen irányba halad a fejlesztés.

Napi Scrum: Scrum Master vezeti, Maximum 15 perces, Mindennap ugyanott és ugyanakkor, Standup – tehát állva.

Sprint Review Meeting: A Csapat és a Scrum Master vesz részt rajta.

Sprint Retrospektív: minden Sprint végén a folyamat javítását célozza. Scrum Master vezeti le. Mi volt jó? Min lehetne javítani? A legfontosabb pontokhoz feladatokat és felelősököt rendelnek.

„Team velocity” – a csapat sebessége: A projekt követés eszköze, mérőszáma. minden iteráció végén a csapat összeadja az adott iterációban ténylegesen befejezett user story-khoz kapcsolódó becsléseket. Az összeg a csapat sebessége (velocity).

Kanban: A Kanban Módszer olyan tervezési, vezetési és javítási eszköz, amely segít a teendőket folyamatukban látni.

A Kanban Board vizuális eszköz, amely több

oszlopból áll. Mindegyik oszlop a

munkafolyamat más és más fázisát jelöli.



Támogató folyamatok

Konfigurációmenedzsment (CM)

Célja a keletkező munkatermékek integritásának biztosítása, felhasználva az alábbiakat:

- Konfiguráció azonosítása
- Konfiguráció ellenőrzése
- Konfiguráció állapotának követése
- Konfigurációs auditok

Tevékenységek:

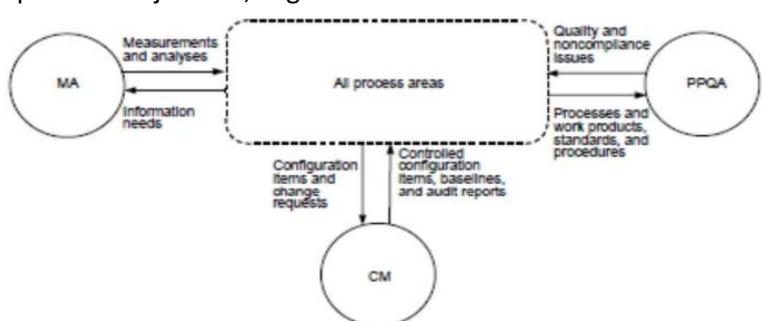
- A kiválasztott munkatermékek konfigurációs elemeinek azonosítása.
- A konfigurációs elemek változásának követése.
- Az alapverziók (baselines) integritásának biztosítása.
- Pontos képet adni a konfiguráció aktuális állapotáról a fejlesztők, végfelhasználók és vevők számára.

Mit kell konfigurációkezelés alá vonni?

Mindent, ami fontos!

CM a CMMI-ben:

- Kettes érettségi szinten.
 - Az összes többi folyamatot támogatja!
1. Alapkonfigurációk létrehozása
 2. Változás követés- és ellenőrzés
 3. Integritás biztosítása



Alapverziók: Az alapverziók biztosítják, a konfigurációs elemek stabil alapokkal rendelkezzenek a továbbfejlesztéshez. Az alapverziókat – amint létrehoztuk őket – hozzáadjuk a konfigurációmenedzsment rendszerhez.

Configuration control board (CCB): A konfigurációs változások kezelésével megbízott csoport.

„Configuration control”: A CM eleme, amely a konfigurációs elemek formális azonosítása után bekövetkező változásokat elemzi, koordinálja, jóváhagyja vagy elutasítja, és a végrehajtott változások implementálást követi.

CM agilis környezetben:

Fontos, mert támogatni kell a gyakori változásokat, gyakori (napi) build-eket, többfélé alapverziót és többfélé munkakörnyezetet stb-t.

Hasznos:

- 1) a CM-et automatizálni
- 2) a CM-et egységes, szabványos szolgáltatásként építsük fel.

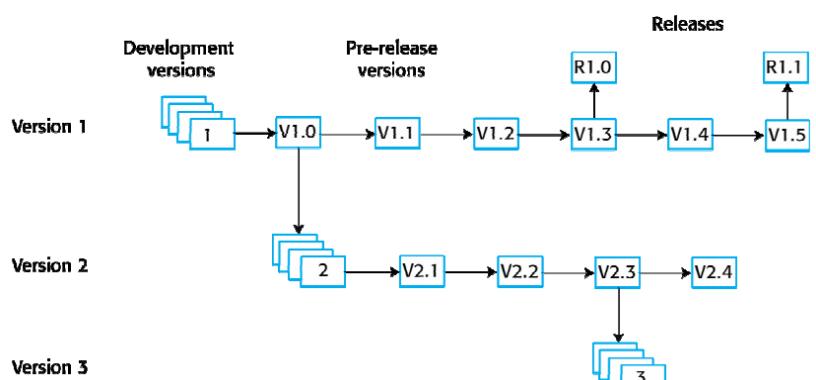
Kollektív tulajdon(lás): Explicit konvenció, miszerint minden csapattag minden kód-file-hoz hozzáférhet és azt, szükség szerint, módosíthatja. Ezt azért teheti, hogy teljesíten egy fejlesztési feladatot, vagy egy hibát kijavítson, vagy javítsa a teljes kódminőséget.

Verziókezelés (VC)

A konfigurációmenedzsment része.

A verziókezelő rendszerek a komponensek különböző tárolt verzióihoz való hozzáférést szabályozzák.

Lehetőleg automatizáljuk!



Kétféle modern verziókezelőrendszeret tartanak számon:

- **Központosított rendszerek:** Egyetlen „master” repository tárolja a készülő szoftver összes verzióját.
Pl: Subversion (SVN)
- **Elosztott rendszerek:** A komponens repository-nak egyszerre több verziója létezik ugyanabban az időpillanatban. Pl: Git

Kockázatmenedzsment

Kockázat: Annak a valószínűsége, hogy előre nem látható esemény fordul elő.

A bizonyosság hiánya arra nézve, hogy egy tevékenység eredménye azonos lesz a tervezettel.

Annak lehetősége, hogy egy tevékenység elvárt és tényleges eredménye között bizonyos különbség merül fel.

A kockázat fajtái: Műszaki, Pénzügyi, Kereskedelmi, Erőforrás, Vállalati...

Kockázatmenedzsment:

- Azoknak a tevékenységeknek az összessége, amelyek elősegítik, hogy egy tevékenység tényleges eredménye a lehető legjobban közelítse meg az elvárt eredményt.
- A kockázat mérséklésére szolgáló intézkedések.
- Tevékenységek:
 - A kockázat azonosítása
 - A kockázat elemzése
 - A kockázati tényezők fontossági sorrendjének megállapítása
 - A kockázat elhárítását célzó intézkedések számbavétele
 - A megfelelő intézkedések kiválasztása, bevezetése
 - Az intézkedések eredményének követése
- Lehetséges, a kockázat elhárítását célzó intézkedések:
 - **Elkerülés:** Ne indítsuk azt a tevékenységet, amelynek kimenetele kétséges egy nyilvánvalóan létező, ható kockázati elem miatt.
 - **Csökkentés:** A tevékenység megkezdése előtt tegyünk lépéseket, hogy az azonosított kockázati elem hatása minimálisra (elfogadható nagyságúra) csökkenjen.
 - **Kompenzálás:** Fogadjuk el a kockázati tényező negatív hatását a tevékenységre, de egyéb tényezőkre figyelve igyekezzünk ezt a negatív hatás elfogadható nagyságrendűre csökkenteni.
 - **Megegyezés:** Tételezzük fel, hogy a kockázati tényező kifejtíti hatását, és készüljünk fel a negatív hatás kezelésére.

Kockázatmenedzsment a CMMI-ben:

A kockázatkezelés célja a potenciális problémák azonosítása, mielőtt azok bekövetkeznének.

A kockázat olyan valami, ami megjelenhet és váratlan vagy előre nem látható kimeneteket okozhat.

1. Kockázatkezelés előkészítése
2. Kockázatok azonosítása és elemzése
3. Kockázatok csökkentése

Minőségmenedzsment

- A fontos minőségi attribútumok értékéről méréssel kell meggyőződniük.
- A mérések egy részét teszteléssel végezzük.
- De: A szoftver minőségét nem elegendő teszteléssel biztosítani! Új hibaelkerülési, hibamegelőzési technikák szükségesek a megváltozott szoftverfejlesztési körülmények miatt!
- A szoftver jó minőségének biztosítása tudatos és folyamatos befektetéssel és a minőség költségének vállalásával!
- A szoftverminőség függ a konkrét helyzettől.
- Minőségi profilt kell meghatározni minden esetben!
- A minőségi profil kialakításakor ismerni kell a szoftverminőség fontos elemeit és a létező megközelítéseket.
- A cég konkrét igényeinek megfelelő elemeket és megközelítéseket kell kiválasztani.
- Garvin szoftverminőség definíciói: Transzcendens, Termék alapú, Felhasználói alapú, Érték alapú, Folyamat alapú
- A szoftverfejlesztő cégeknek **bizonyíthatóan** jó minőségű szoftvert kell előállítaniuk; különben nem tudnak a piacra maradni.
- A szoftver auditálása:
 - Egy audit egy szisztematikus bizonyíték gyűjtő folyamat.
 - Szoftverfejlesztő cégeknél végzett auditok sajátosságai: Szoftverfejlesztésről szól. Kapcsolódhat a szoftver termékhez vagy / és szoftver folyamatokhoz.
 - Mintavételek – a minta legyen reprezentatív!
 - Auditok eszközei: mintavétel, interjúk, dokumentumok ellenőrzése
- Szoftver termék minőségi követelmények vonatkozhatnak: Kódra + dokumentumokra + adatokra +...
- Ne feledjük! A szoftver termék minősége sokkal több a kód minőségénél!!!!
- PPQA - Folyamat és termék minőségbiztosítás a CMMI-ben.

Mérés és elemzés

Mérés:

- A teljes szoftverfejlesztés ellenőrzésének eszköze.
- Az a folyamat, amelynek során a valós világ elemeinek attribútumaihoz számokat vagy szimbólumokat rendelünk, abból a célból, hogy valamilyen, jól meghatározott szabály alapján leírjuk, jellemezzük őket.
- Valamilyen célja van.
- Eredményes méréshez:
 - Érteni kell, hogy milyen elem (objektum) milyen vonatkozását (attribútumát) mérjük, és miért.
 - A sok lehetőség közül ki kell tudnunk választani az adott célnak megfelelőt.
 - Jó mérőszámot és méri mi módszert kell választani.
 - A mérés eredményét elemzni kell és fel kell használni
- **Direkt mérés:** egy attribútum vagy entitás mérése nem feltételezi más attribútumok vagy entitások bevonását. (pl. hosszúság)
Direkt mérőszámok (példák): forráskód hossza, tesztelési folyamat időtartama, a tesztelés során megtalált hibák száma, egy programozó által a projekten töltött idő
- **Indirekt mérés:** egy attribútum vagy entitás mérése csak további entitások vagy attribútumok bevonásával lehetséges (pl. sűrűség = tömeg/térfogat)
Indirekt mérőszámok (példák): programozó teljesítménye: megírt LOC / ember-hónap, hibasűrűség egy modulban: hibák száma / modul, mérete hibamegtalálási hatékonyság: megtalált hibák száma / összes hibák száma, követelmények stabilitása: kezdeti követelmények száma / összes köv. sz., tesztelési hatékonysági arány: tesztelt elemek száma / összes elem száma, „hulladék-arány”: a hibajavítás ráfordítása / a projekt teljes ráfordítása
- Funkciót pont számolás: A szoftvertermék komplexitásának mérésére és becslésére.
- A projektmenedzsment folyamatok és erőforrások mérése
 - Projekthez kapcsolódó mérőszámok, projektkövetés során mérhetők.
 - Belső folyamat-attribútumok: a projektre fordított idő, a szükséges erőforrások és a költségek. Ezek mérése egyszerű feljegyzést jelent.
 - Külső folyamat-attribútumok: pl: ellenőrizhetőség, megfigyelhetőség, stabilitás ...
 - Direkt mérések: Felhasznált költségek (ACWP), Adott időszakra ütemezett munkára tervezett költség (BCWS), Megvalósult érték = Elvégzett munkára tervezett költség (BCWP)
 - Indirekt mérések: Elvégzett munka alapján számolt költségeltérés (CV= BCWP - ACWP), Elvégzett munka és eltelt idő alapján számolt költségeltérés (SV=BCWP-BCWS), Költség szerinti teljesítmény index (CPI=BCWP/ACWP), Időzítés szerinti teljesítmény index (SPI=BCWP/BCWS)
- Mérés és elemzés (MA) a CMMI-ben:
 - A mérés és elemzés célja olyan méri mi ismerettár (kelléktár) fejlesztése és fenntartása, mely a menedzsment információs igényeit kielégíti.
 - Quality Improvement Paradigm (QIP): Folyamatos javításra koncentrál. Az egyes projektek tapasztalatát elemzi, „csomagolja” további projektekben való felhasználhatóság szempontjából.
 - Mérés agilis környezetben: folyamatos tevékenység