Programozási Technológiák

Áttekintés

Rendszerfejlesztés lépései:

1. **Követelmények feltárása.** Amikor megfogalmazzuk azt, hogy a kifejlesztendő információs rendszernek (egyszerűbben fogalmazva szoftvernek) milyen körülmények között és mit kell tudnia. Tehát meghatározzuk a követelményeket.
2. **Elemzés (analízis, kalkulus).** Amikor a követelményeket megvizsgáljuk, elemezzük, csoportosítjuk olyan szempontból és azért, hogy majd a szoftvert megfelelően tudjuk kifejleszteni.
3. **Architekturális tervezés (logikai, magas szintű tervezés).** Az architektúrális tervezés a szoftver struktúrájának a kialakítására vonatkozik (lehetőleg mindenféle platformtól, hardvertől, szoftvertől és egyebektől függetlenül).
4. **Tervezés (fizikai, alacsony szintű tervezés).** A tervezésben viszont már figyelembe vesszük ezeket. Tehát a struktúrának a felépítése figyelembe véve azt, hogy a következő fázisban majd hogyan és mit csinálunk.
5. **Implementálás.** Kódolás, megvalósítás, amelynek nem elhanyagolható része a tesztelés. Ennek a végén készül el maga a szoftver.
6. **Üzembe helyezés (bevezetés)**
7. **Üzemeltetés.** Futtatjuk a szoftvert.
8. **Karbantartás.** Ha hibás a szoftver, akkor a hibákat kijavítjuk.
9. **A szoftver fejlődik (Evolúció).** Gyakran a karbantartás helyett. A szoftver módosul, fejlődik, mert változnak a gazdasági, jogi, stb. viszonyok, amelyek között használjuk.
10. **Üzemen kívül helyezés.** Általában nem szoktak hangsúlyt fektetni rá.

(Mi a logikai tervezéssel, fizikai tervezéssel és implementálással foglalkozunk.)

Szoftver alatt a **program**, a **dokumentáció** és az **adatok együttesét** értjük! Különös tekintettel a dokumentációra. A dokumentáció a teljes életciklust végigkíséri.

A szoftver egy termék. Egyrészt ugyanolyan termék, mint bármely ipari termék, másrészt van egy alapvető különbség: ez egy szellemi termék, ennek az összes hátrányával. Speciális, mert szellemi termék, ez alapvetően megkülönbözteti más ipari termékektől.

A szoftvert, mint terméket, elő kell állítani. Projektben lehet előállítani. Részben azonos jellemzőkkel rendelkeznek, részben speciálisak. A minőség alapvető szerepet játszik. A projekt idő, pénz és egyéb erőforrás.

Szabványok

Szabványok fajtái:

- **„de jure”.** Amelyeket mindenfajta szabványügyi testületek, hivatalok, intézetek definiálnak, formalizálnak. Két legnagyobb: **ISO ANSI**. Az ezek által előállított szabványokat mindenütt, mindenki elfogad. Legfeljebb némi módosítással. Senki nem vitatja őket.
  Problémái:
  Elviekben angolul vannak, nehéz olvasni őket, formalizáltak és a szóhasználat sokkal bonyolultabb, mint a jogszabályokban. Általában nagy terjedelműek.

- **„de facto”.** Nagyobb szerepet játszanak. Ezek ipari szabványok. Egy-egy **szakterületnek a**

szabványai, nem általánosak. Az adott szakterület vezető cégei által delegált emberek fogalmazzák meg őket. Az adott szakterület legjobb gyakorlatát gyűjtik össze. Az informatikában az egyik legnagyobb szabványgyártó konzorcium az **OMG** (Nagyon sok OMG szabvány van. Közel 2000 tagja van. http://www.omg.org). A másik a **W3C**. Az OMG a nyílt, elosztott objektum orientált rendszerek ipari szabványait állítja elő. Az ipari szabványok azért lényegesek, mert az adott szakterület minden jelentős tényezője hozzá rakja a saját magáét. Szabvány módon születnek. Fejlődési trendeket jelölnek ki.

- **Piaci „de azért is”.** Szakterületekhez kötődik. Az adott szakterületen van egy, vagy több **nagyon erős cég** és, amit a cég csinál, azt kell csinálnia a többi cégnek is. Van egy fajta gyakorlat, és mindenki ahhoz igazodik.
- **Vállalati.** A legszűkebb. Egy-egy adott cégen, intézményen, **vállalaton belül** saját szabványok léteznek, amelyek igazodnak az előzőekhez, abból származnak, testre szabott szabványok. Bizonyos nagyságrenden felül minden vállalatnak lennie kell vállalati szabványának. Ez a legflexibilisebb szabvány.

A szoftverkrízis egyrészt létrehozott sok új informatikai eszközt, paradigmát, másrészt létrejöttek a programozási módszertanok (moduláris, struktúrált, OO).

Az egész informatikai fejlődés arról szól, hogy olyan paradigmák, eszközök jelennek meg, melyekben egyértelmű cél az absztrakciós szint növelése, modellezés. Egy másik fontos fogalom az újrafelhasználás. Ezek köré formálódik a ma informatikája.

Minőségbiztosítási szabványok

Nagy szabványgyártó cégek: ISO, ANSI, IEEE, (NATO)
ISO9000 az általános folyamatszabvány. Speciálisan informatikai az ISO9001.
Minden termékre alkalmazható szabvány. Amit minden cég saját magára szabhat, és létrehozza a saját minőségi kézikönyvét. Ez a minőségbiztosítás. Ezeket szabja rá a projektekre, és így történik a minőség-ellenőrzés. Gyakorlatilag egy saját szervezeti minőségkezelési folyamatot definiál.

(ábra)

Vannak auditor cégek amelyek, minőségi tanúsítványokat adnak ki. Az auditáló cégek által adott tanúsítvány arról ad tanúbizonyságot, hogy az adott cégnek van minőségi kézikönyve, van minőségtervezése, és minőség-ellenőrzése.

Dokumentációs szabványok

Minden cég saját dokumentációs szabványt dolgoz ki. A minőségkezelés lényeges adminisztratív rész. Ehhez a csoporthoz 3 szabvány tartozik:

1. **Dokumentációs folyamatszabványok** - Követni kell a dokumentáció előállítása során.
2. **Dokumentumszabványok** - Hogyan nézzen ki egy doksi. Ezen belül szokás ezeket megadni:
    a. **Dokumentum azonosítás szabványai**, ahhoz hogy hivatkozhatók, kezelhetők legyenek a dokumentumok.
    b. **Szerkezetre vonatkozó szabványok**, melyek meghatározzák milyen részekből álljanak az egyes dokumentumok.
    c. **Kinézetre vonatkozó szabványok**, pl.: logo, saját font stb.
3. **Dokumentumcsere szabványok** - Hogyan kell végrehajtani a változtatást a dokumentumon, és hogyan jelöljük a változást.

A dokumentumokat jóvá kell hagyni!

UML (Unified Modelling Language)

Grafikus modellező nyelv. **OMG szabvány.**
Az OO a 80-as években rendszerfejlesztési módszertanná vált. (90 -es évek elejére több, mint 100.) A 80-as évek elején 3 nagy irányzat: Booch-Jacobson-Rumbaugh. Más-más dolgokra helyezik a hangsúlyt. A strukturált módszertanokból vesznek át bizonyos elemeket, és azok mellé raknak OO elemeket. A 90-es évek elején összefogtak, összevonták a módszertanaikat. Egy egységes módszertan kidolgozását tűzték ki célul. 95-ben megjelent az UML 0.9. 97-ben OMG szabványként megjelenik az UML 1.1. Ez az első szabvány. 2001-ben az 1.4, majd az 1.5, innentől kezdve kezd romlani. 2005-ben megjelenik az UML 2.0, majd 2007-ben az **UML 2.1** (Ez az az OMG szabvány, amit ma UML alatt értünk).
Az **UML** legnagyobb előnye az, hogy **mindenkitől független**, nem kötődik semmilyen informatikai céghez. **Mindenki elfogadja**, mert nem kötődik a másikhoz. Időben jött, a 90-es évek közepén megszülettek az OO szabványok, megszületik a web, az OO szemlélet, mint paradigma teljesen általánossá válik. Az **UML** azonnal belép **a követelmény-feltárásnál, követelmény-elemzésnél és tovább folytatódik az implementálás vonatkozásában.** Illetve, **mint modell, a tervezésnél játssza a legalapvetőbb szerepet.**
Az **OMG megalkot egy saját formalizmust**, egy szabvány definíciós formalizmust, minden szabványát ennek a formalizmusnak a segítségével definiálja. Ez egy meta-cirkuláris szabvány, **alapja az UML**. Ahhoz, hogy ezt a formalizmust tudjam, ismerni kell az UML-t, de az is ebben a szabványban van definiálva. A **szabványdefiníció csúcsán a MOF** (Meta Object Facility) áll. Ez az egész szabványdefiníciós formalizmus **egy 4 szintű architekturális minta.**

Az UML, mint szabvány legfontosabb tervezési elvei a következők:

- **Modularitás.** A nyelvi konstrukciókat csomagokba szervezi.
- **Rétegzettség.** A különböző szintek példányai a magasabb szinteknek (4 szint).
- **Particionálás.** A csomagokat különböző partíciókra osztja a szabvány.
- **Kiterjeszthetőség.** UML profilok hozhatók létre. UML, mint nyelv, kiterjeszthető platformspecifikus profilok felé. Profilok hozhatók létre a szakterületre.
- **Újrafelhasználhatóság.** Az UML jelen pillanatban létező elemeit újra felhasználhatom. **CWM** (Common Warehouse Metamodel - Adattárház Metamodel) szabvány.

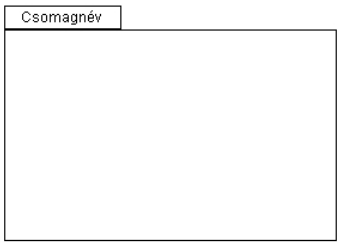
Egy nyelv megalkotásához a következők kellenek:

- **Absztrakt szintaxis:** definiálja a nyelv konstrukcióját jelöléstől függetlenül.
- **Konkrét szintaxis:** ez a jelölés.
- **Statikus szemantika:** megmondja, hogy egy nyelvi konstrukció példányai hogyan kapcsolódnak más példányokhoz.
- **Dinamukis szemantika:** jól formált konstrukciók jelentését definiálja.

Az UML a 4 szintű formalizmussal zárt. A szabvány konstrukción kívül semmi más nem szükséges a definícióhoz és bármilyen modell leírásához. A szemantikát természetes nyelven (angol) adja meg, kivéve az OCL.

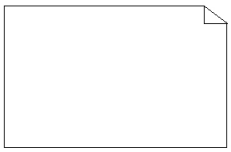
A diagramok egy része a viselkedés leírását, más részük a struktúra leírását adja.

Diagramok:
Csomag:



A csomag tulajdonképpen egy névtér. Típusokat és csomagokat tartalmaz. Tehát az UML típus-alapú rendszer. A hatáskör elhatárolására csomagokat alkalmaz. A csomagnévvel való minősítés **::** .

Megjegyzés:



Bármely konstrukcióelemhez fűzhetünk megjegyzést.

Sztereotípus (sztereotípia):

Ezt is adhatunk bármely elemhez. Vagy pontosítja az adott elemet, vagy a kiterjesztésében játszik szerepet, vagy egy új modellelemet hoz létre.

<<sztereotípus_neve>>

A szabványban sok beépített sztereotípia van.

Legfelső szintű csomag:

<<system>>.

Függőségek:

Egy csomagon belül elhelyezett alcsomagok között értelmezettek. A függőség jele:

<----

Pl.: <----<<import>> csomagok közötti publikus import az adott irányban.

Osztály:

(A legfontosabb.) Az OO paradigma azt mondja, hogy a való világ modellezése közben a világot objektumokra robbantjuk szét és a követelmény-feltárás és elemzés feladata az, hogy az adott problémához tartozó objektumokat kiderítsük, majd az elemzés során ezeket, az objektumokat kategorizáljuk, osztályokba soroljuk. Az OO értelemben vett osztályokba soroljuk és ezeket, az osztályokat formalizáljuk az osztálydiagram segítségével.

Az UML vonatkozásában az osztálynak 4 fajta közelítése lehet:

a) **Fogalom**. A követelmény-feltárásnál alapvető, amikor egy szakterületi fogalom absztrakciója jelenik meg egy osztályban. (pl. tanulmányi rendszernél a hallgató.).
b) **Típus**. Az osztály, mint absztrakt adattípus. Az elemzésnél.
c) Az **osztály, mint objektumoknak egy halmaza**. Ez az adatbázisokhoz kötődő objektumorientált adatmodellezés környékén egy közelítés. Tipikusan ebben a közelítésben, amikor halmazként tekintjük az osztályt, azt mondjuk, hogy van egy szuperosztály, meg egy alosztály. Az alosztály, mint halmaz, minden eleme eleme a szuperosztálynak, mint halmaznak.
d) Az **osztály, mint implementáció**. Mint egy Java osztály, vagy egy C# osztály.

Az UML mind a 4-et segíti, lehetővé teszi.

Osztálydiagram:

Név: Nagybetűvel kezdődik, középre igazított és a környékén sztereotípiák, megjegyzések lehetnek.

Attribútumok:

név [:típus] [=kezdőérték]

Adhatok neki típust, kezdőértéket, de egyik sem kötelező. Az attribútumok neve kisbetűvel kezdődik, a több szóból álló nevek második, harmadik stb. szavának kezdőbetűje nagybetű. A származtatott attribútumok neve előtt egy / áll. Az osztályszintű attribútumokat aláhúzzuk.

Műveletek: pontosabban a műveletek szignatúrája.

név (paraméterek) [:típus]

A névre ugyanaz vonatkozik, mint az attribútumok nevére. A zárójel a névhez tartozik, a paramétereket vessző választja el egymástól, ha többen vannak. Ha van típus, akkor függvény-jellegű művelet, ha nincs típus, akkor eljárás jellegű művelet.
Egy paraméter a következőképpen néz ki:

[mód] név : típus [=kezdőérték] (paraméter)

A mód lehet IN=csak olvasható, OUT=csak írható, INOUT=írható/olvasható, RETURN=visszatérési paraméter (ek).

Nagyon gyakran, ha nem lényegesek a részletek, vagy nagyvonalú elemzésről és tervezésről van szó, az osztálydiagram úgy jelenik meg, hogy csak a név van benne, nincsenek attribútumok és műveletek.

Bezárás, láthatóság:

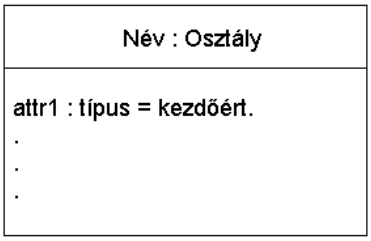
Vagy minden attribútum és művelet neve előtt megadjuk, vagy bezárási szekciókat hozunk létre.

+ public
 # protected
- private
~ (csomagláthatóság)

Megszorítás:

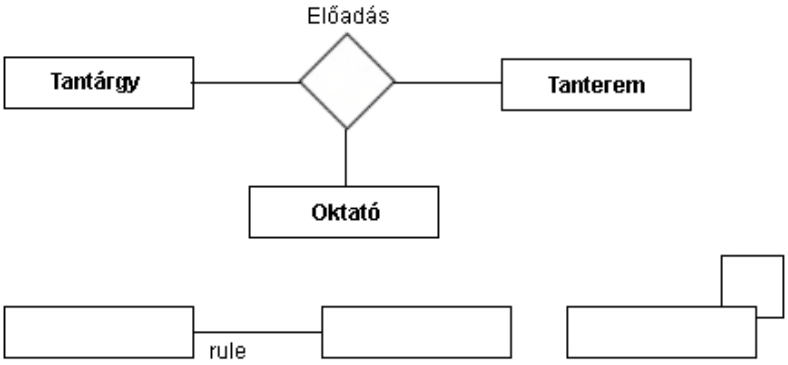
Bármely elem mellett {} között megadhatok megszorítást. A megszorításhoz kapcsolódik az OCL (Object Language Constraint). Adatbázis értelemben vett megszorítás.

Objektumdiagram (példánydiagram):

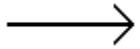
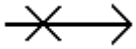


Példány: megnevezem a példányt, a példányosító osztályt.
Attribútum = érték: ezzel adom meg a kezdőállapotot.

Asszociáció:



Osztályok közötti strukturális viszonyt modellezünk vele. (Ez egy kapcsolattípus.) Az asszociáció tetszőleges fokú, de minimum 2. Jelölése az osztályok között folytonos vonal. Nincs kizárva, hogy az asszociáció ugyanazt az osztályt kösse össze önmagával. Van két vége. Az asszociáció végei megnevezhetőek szerepkörök segítségével. Létezik az asszociációnak számossága. Két beépített megszorítás, ami az asszociációhoz kötődik:
{ordered}, {sorted}.
sorted: a kapcsolatban a példányok rendezettek.
ordered: meghatározott sorrendben érhetők el, de nem mond semmit erről a sorrendről.
rule**:** szerepkör. Egy kapcsolat mindkét végén elhelyezhető.

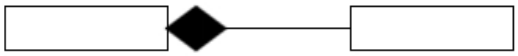
Az asszociációnál a folytonos vonalon lehet jelölni a navigálhatóságot. Ez vagy egy nyitott végű nyíl  a cég példányai hivatkozhatják az alkalmazott példányokat, fordítva nem mond semmit. Letiltani a navigálhatóságot így lehet:  a cég példányai hivatkozhatják az alkalmazott példányokat, fordítva ez nem igaz.

Aggregáció:

Osztályok közötti speciális viszony. Szokás ezt egész-rész viszonynak hívni. Aszimmetrikus viszony, nem kommutatív, de tranzitív. Az egész oldali attribútumok és műveletek megjelennek a rész oldalon. Az aggregáció a legvitatottabb UML fogalom.
(ábra projekt,alprojekt…)

Kompozíció:

Az aggregációra rárak egy kezelési szemantikát. A rész nem élheti túl az egészet.



Öröklődés (UML-ben általánosítás, pontosítás):

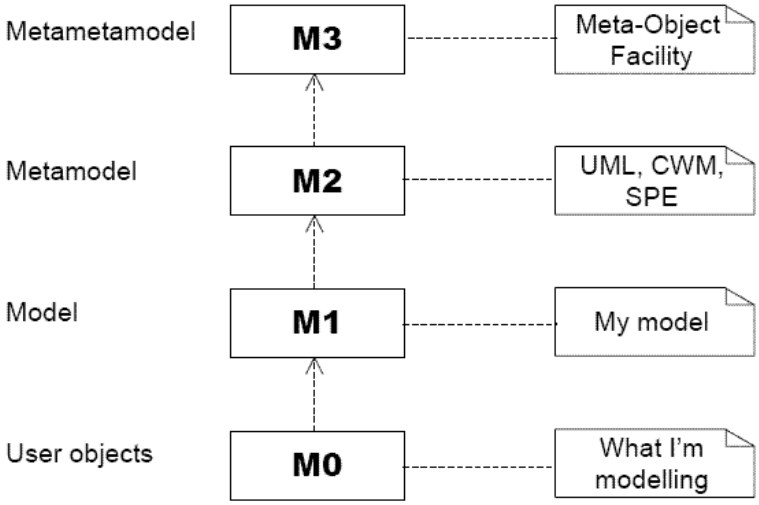
Klasszikus öröklődés, többszörös. A helyettesíthetőség elvén alapul. A 2.0, 2.1-es UML az egységesség elvét vallja. Tehát minden nyelvi konstrukció objektum, minden nyelvi konstrukció egy osztálynak példánya.

Az UML szabványról:

Két alap eszközrendszer van, az **infrastruktúra** és a **szuperstruktúra**. Az infrastruktúra könyvtár az, amelyik a nyelvi alapeszközöket tartalmazza, ezeknek az alap szintaxisát írja le.
(ábra Az SQL92 create table utasítása)

A való világot modellezzük, a modellt meg kezelni kell. Az OO paradigma együtt kezeli a struktúrát és a viselkedést. Ahhoz, hogy a modellt kezelni tudjuk, kell egy kezelő nyelv. Mindenféle modellnek kell egy kezelő nyelv. A valós világ modellezésénél megjelenhetnek a következő egyedtípusok (OO-ban osztályok): Hallgató, Oktató, Tantárgy stb., amelyeknek a példányai a tényleges egyedelőfordulások. És mondjuk a modell kezelő nyelve, lehet a Java, vagy a C#, vagy az SQL stb.
A kezelő nyelv kezeli az osztályokat és kezeli a példányokat is. De van egy alapvető különbség. Az egyedtípusok, az osztályok, azok modellbeli eszközök, a modell eszközei. És a konkrét példányok az egyedek. És lényeges, hogy absztrakciók. A modell elemei absztrakciók. Ezeket az absztakciókat le kell tudni írni. A példányok és az egyedek viszont konkrét dolgok. Csak ezek a konkrét dolgok. És, amikor futtatjuk a Java, C#, vagy SQL scripteket, akkor léteznek, tehát futási idejű dolgok.
A kezelő nyelvet is definiálni kell, formalizálni kell. Van szintaxisa és szemantikája. Ezeket valamilyen módon meg kell adni. Ezek a nyelvek szöveges nyelvek, szöveges eszközrendszerrel rendelkezik.

Az **UML grafikus nyelv**. Új formalizmust kerestek és új módszert találtak. Ugyanis az OMG azt mondja, hogy a kezelő nyelvet ne egy szintaxist leíró formalizmus segítségével definiáljuk, hanem egy metamodell segítségével. Tehát adjunk egy modellt a kezelő nyelv fogalmainak a megadásához. Ez lesz a metamodell. A modellt leíró kezelő nyelv elemei, azok a metamodell osztályainak a példányai. Tehát a metamodellben osztályokat definiálok, és azok példányai lesznek a kezelő nyelv elemei. A közelítés teljesen más, nem nyelvet definiálok, hanem metamodellt definiálok.



Az OMG a 4 szintet M0, M1, M2, M3-nak nevezi.
M0:(rendszerszint, példányszint, konkrét szint) Itt vannak a konkrét példányok. Ezek példányai a megfelelő modellben definiált osztályoknak.
M1:(modellszint) Azok az osztályok, melynek a példányai az M0 szinten vannak.
M2:(metamodell) Ezeknek a példányai a modell szintű fogalmak.
M3:(MOF) Meta Object Facility. Önmagával van definiálva. Egyetlen MOF létezik. És az előbbi metaosztályok mind MOF osztály példányaként jönnek létre.

Minták (Patterns)

A minták az elmúlt 15 évben kapnak egyre nagyobb szerepet. A minta egy olyan általános megoldás, amely működött a múltban és **újrafelhasználható** módon működni fog a jövőben is. A minták receptek. Mint a receptek általában, a minták is vagy jól használhatók, vagy rosszul. A minták a GoF-nak becézett 4 ember könyvében jelentek meg először. Ők írták le szisztematikusan a mintákat.

Egy minta a következő összetevőkkel mindenféleképpen rendelkezik:

- **Név**.
- **Probléma**, amely probléma megoldására született.
- **Környezet**, amiben használható.
- **Kényszerek**, feltételek, amelyek mellett használható a minta.
- **Megoldás** a problémára, amit a minta kezel.
- **Alkalmazási példák.**

Az informatikán belül a mintákat a következő féleképpen szokás csoportosítani:

- **Analisis** (elemzési minták)
- **Architecture styles** (architekturális minták, stílusok)
- **Design** (tervezési minták)
- **Programming idioms** (programozási idiómák)
- **Process** (folyamat minták)
- **Antipatterns** (ellenminták)

Az első 4 termék minta, implementációs minta.
Nincs éles határ. Ezek az osztályok átfedik egymást.

Elemzési minták

Az elemzési minták **szakterület specifikusak**. Az adott szakterület alapvető szakterületi absztrakcióit tudjuk velük kezelni.

Architekturális minták

Az architekturális minták a teljes rendszerre és alrendszerekre vonatkoznak. Tehát **nagyon magas absztrakciós szint**ű minták. A **MOF** egy 4 szintű speciális architekturális minta.

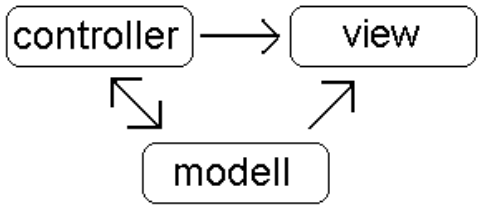
Tervezési minták

A tervezési minták **közepes absztrakciós szint**tel rendelkező minták. Ezek váltak először híressé. Sok tervezési minta van, különböző osztályokba vannak sorolva.

A tervezési minta leírására szolgáló elemek:

- **Név** és osztályba sorolás.
- A **probléma**, elérendő cél ismertetése.
- **Szinonima** nevek.
- **Motiváció**. Mi indokolja, hogy egy ilyen mintával egyáltalán foglalkozzunk?
- **Alkalmazhatóság**. Azok a területek, ahol újrafelhasználható a minta.
- **Struktúra**.
- **Összetevők**.
- **Együttműködés** más mintákkal.
- **Következmények** az alkalmazásra vonatkozóan.
- **Implementáció**.
- **Példakód**.
- **Példa** a használatra. Esettanulmány jellegű.
- **Kapcsolódó minták**.

A leghíresebb tervezési minta a Smalltalk környékén kialakult **MVC (Model View Controller)**. (http://java.sun.com/javaee/5/docs/tutorial/doc/bnahb.html)



Ebből később kialakul a klasszikus 3 rétegű alkalmazás. Az MVC szerint szét kell választani az adatkezelést (model), szét kell választani a megjelenítést (view), és szét kell választani a vezérlést (control). És ebből lesz ez a bizonyos 3 rétegű alkalmazás. Mai terminológiával: adat réteg, üzleti logika réteg, prezentációs réteg.

A tervezési minták nagy része az objektumok közötti kommunikációról szól.
Általában a mintákat felfedezik. A másik közelítés szerint józan paraszti ész.

A tervezési mintáknak 3 nagy kategóriája van:

- **Létrehozó minták:** Azt mondják meg, hogy hogyan kell végrehajtani egy példányosítást.

Magyarul legyártanak számunkra objektumokat. Nem én példányosítok, hanem a minta alapján legyártatom az objektumokat.
- **Strukturális minták:** Objektumcsoportok kezelésére vonatkoznak.
- **Kommunikációs minták, viselkedési minták:** Az objektumok közötti kommunikációt, a vezérlés legjobb gyakorlatát írják le.

Az OO világ tervezési mintái azt szolgálják, hogy az OO világban a legszigorúbb módon megvalósítsuk az ADT szemléletet. Az osztályok minél lazábban kapcsolódjanak, az öröklődést komolyan vegyük, az újrafelhasználás öröklődés alapú legyen.
Tervezésnél az ADT-re és az öröklődésre kell koncentrálni.

Ökölszabály az absztrakció. Tervezésnél interfészek vannak. Magyarul az öröklődés viselkedési és nem strukturális öröklődés, amiben a tervezésnél gondolkodni kell. Absztrakt viselkedést definiáló interfészek és a közöttük levő öröklődési kapcsolatok, vannak körülfogva. Tervezésnél absztrakt osztályban gondolkozunk, ha az interfész túl absztrakt. Legvégső soron konkrét osztály.

Gyakran az OO alkalmazásfejlesztésnél, rendszerfejlesztésnél **az öröklődés helyett objektumkompozíciót kell használni**. Magyarul konténereket, kollekciókat kell használni. Ugyanis ezen imperatív OO paradigmában az öröklődéssel van egy probléma. Az öröklődés megzavarható a bezárással, tehát kicsit ellentmondanak egymásnak. Az imperatív OO paradigma egyértelműen a helyettesíthetőségen épül fel, amihez kell a láthatatlan öröklődés.

A tervezési minták ma megjelennek fejlesztői környezetekben beépített módon.

Programozási idiómák

Legelőször a programozási idiómák jelentek meg. Ezek programnyelvhez kötődnek. Totálisan konkrétak.

Egy adott programnyelven megtanulni programozni a következőket jelenti:
- Meg kell ismerni az adott nyelv **szintaktikáját és szemantikáját** (3 nap)
- Meg kell ismerni egy **fejlesztőkörnyezet** eszközrendszerét (3 év)
- El kell sajátítani az adott programnyelv **programozási stílusát** (30 év)

A programozási minták konkrét nyelvhez kötődő, **nagyon alacsony szintű** minták, amelyek a jó programozási stílust szolgálják.

A jó programozási stílushoz hozzátartozik:
- **Elnevezés**i konvenciók.
- Forrásszöveg **formázás**a.
- Az adott **nyelv elem**einek a jó használata.

Egy adott problémát egy adott nyelven sokféleképpen meg lehet oldani.
Pl.: a és b értékének felcserélése:

```
a = a + b
b = a – b
a = a – b
```

Az **idiómák a legjobb gyakorlatot** gyűjtik össze. Ezek azok a fogások, melyek gyakran nincsenek még leírva sem. Bizonyos idiómákat bizonyos cégek **abszolút belső titokban** kezelnek. Különösen igaz ez az adatbázis programozás területén.
Minden nyelv mögött ott vannak a programozási idiómák.
(Java-hoz egyik legjobb könyv: Bloch – Effective Java Programming Language Guide)

Folyamatminták

A folyamatminták a rendszer előállításának folyamatára vonatkoznak.

Antiminták (http://www.sourcemaking.com/antipatterns)

Az antiminták, szemben a mintákkal, a legrosszabb gyakorlatot gyűjtik össze. Olyan problémákat, hibás szituációkat, nem megfelelő gyakorlatot, amit el kell kerülni. Az antiminták olyan minták, melyek megadják a megoldást is, tehát a hiba javítását, elkerülését, megoldását.

A következő kérdésekre lehet választ adni az antiminták segítségével:

- Melyek a legáltalánosabb szoftvertervezési, szoftver fejlesztése során előforduló hibák?
- Hogyan ismerhetők ezek föl?
- Hogyan javíthatók?
- Hogyan lehet felismerni, hogy egy szoftver projekt zsákutcába jutott, és hogyan lehet onnan kihozni?
- Hogyan lehet felfedezni azt, hogy a szoftvergyártó átvert bennünket, vagyis nem azt adta, amit ígért?
- Hogyan lehet eldönteni, hogy a legújabb termék, szabvány, technológia választ ad a jelenlegi problémánkra?
- Melyek az újrafelhasználás veszélyei, problémái?

Egy minta nagyon hamar antimintává válhat, ha az alkalmazás feltételei nem megfelelőek!

Antiminták 3 nagy kategóriája:

1. **Folyamatok és az emberek menedzselése**.
2. **Architekturális** ellenminták.
3. **Implementációs** ellenminták**.**

Az ellenmintáknál a probléma megoldását hívjuk refactoringnek. Legalábbis az implementációs ellenmintáknál a jó megoldást a refactoring segítségével érjük el.

Blob:

Mikor áll elő?

- Akkor áll elő, ha egy programot úgy írunk meg, hogy egy **osztály kizárólagosan uralja a vezérlést** (blob osztály), és hozzá kapcsolódnak kisméretű osztályok, amelyek az adatokat kezelik, tartalmazzák.
- Ha egy EO módon megírt kódot ültetünk át OO környezetbe refactoring nélkül.

Mik a jellemzői?

- Egy osztály, ami **50-nél több tag**ot tartalmaz már blob osztály.
- Ha egymástól **független metódusok és attribútumok** jelennek meg az osztályban. Vagyis olyan atribútumok vannak, amelyeket nem hivatkozik metódus.
- A programban **egy aktív és sok passzív osztály** jelenik meg.

Miért áll elő?

- Az OO paradigma **félreértelmezése**, félrealkalmazása okán. Tehát az OO világban EO módon programozok. A főprogramot megírom egy blob osztályban.

- **Architekturális tervezési probléma**. Rosszul tervezem meg az architektúráját, mert a felelősséget bezsúfolom egy osztályba.
- Lehet jó a tervezés, vagyis az architektúra, de **rossz az implementáció**.

Következmények:
- A rendszer nehezen módosítható.
- A funkcionalitásokért felelős szoftverrészt nehéz megtalálni.
- Tipikusan nem újrafelhasználható.
- Egy blob osztály sok rendszererőforrást köt le, a futási hatékonyság gyenge.

Megoldás:
Refactoring, vagyis a bezsúfolt funkcionalitást, viselkedésmódot szét kell tagolni! A komplexitást radikálisan csökkenteni kell!
Refactoring lépései:
1. A széttagolásnál meg kell határozni azokat az attributum csoportokat, és metódusokat, amelyek összetartoznak, amelyeket egységbe kell zárni! Vagyis ezeket az attributum, metódus csoportokat ki kell emelni, és egy külön osztályba helyezni!
2. Meg kell szüntetni a redundáns kapcsolatokat, viszonyokat! Általában meg kell szüntetni a laza kapcsolatokat!
3. Az eddig lépések során megstrukturáltuk az OO szemléletnek megfelelően az osztályszerkezetet. Megszűntettük a nagy komplex osztályt, széttagoltuk a funkcionalitást, és megvalósítottuk az egységbezárást. Ezután meg kell csinálni az absztrakciót! Vagyis föl kell építeni egy osztályhierarchiát, egy öröklődési hierarchiát!

Fogalmak:
- **Sima-, vagy előretervezés** (Forward engineering). Amikor a **terv hamarabb** készül el, mint a szoftver.
- **Visszatervezés** (Reverse engineering). Egy meglévő **kód alapján** készítem el a **tervet**.
- **Ősrendszerek**. Jól működnek viszont régen tervezték őket, és gyengén dokumentáltak. Problémájuk még, hogy gyakran hardver és operációsrendszer függőek. Viszont mivel a **funkcionalitásuk tökéletes érdemes megtartani őket**. **Ki is lehet** dobni. **Vagy migráljuk őket** (újra legyártjuk), de ehhez tudni kell, hogy mit csinál. Itt jön a visszatervezés, vagyis a kód alapján készítünk egy tervet, aminek a segítségével újra felépíthetjük a rendszert új technológiával. **SOA**: az ősrendszerek megmaradnak, és fölé húznak egy új architektúrát.
- **Újratevezés** (Reengineering). A rendszert **újratervezem a tervek és az implementáció ismeretében**.

Refactoring, újrastruktúrálás (http://www.refactoring.com)

Lényege a következő: Adva van egy implementált szoftver, ekkor a szoftvert változtatjuk meg úgy, hogy a változtatás érintetlenül hagyja a kód külső viselkedését, vagyis a funkcionalitás nem változik semmit, de a kód struktúrája igen. Tehát a refactoring egy kód újrastrukturálás, szisztematikus kódtisztítás. Természetesen, ha újrastrukturálom a kódot, az visszahat a tervre, de ez nem egy újratervezés, hanem egy tervjavítás a funkciók megtartása mellett.

(ábra. lsd. Refactoring Improving the Design of Existing Code - Fowler-Beck-Brant-Opdyke-Roberts.chm First Example)

Újrastruktúrálás főbb szabályai:
- Mindig egy **teszt**tel kell kezdeni, és azzal is folytatni. Mielőtt újrastrukturálunk egy kódot, le kell futtatnunk olyan teszteket, amelyeknél pontosan ismerjük az outputot és inputot. Ez egy olyan teszt, **ami a funkció megtartását szolgálja**, tehát nem egy hiányossági tesztelés.

Minden refactorizálási lépés után tesztelünk.

- **Csökkentsük a metódusok komplexitását**. Ehhez a metóduson belül meg kell találni a kód logikai határait. A rövidebb metódus jobban tesztelhető, változtatható, megérthető. A változtatás során a funkcionalitást meg kell tartani, és nem szabad hibákat rakni a kódba. Külön figyelemmel kell lenni a paraméterekre és a lokális változókra.

- Fontos, hogy minél **kevesebb paraméterrel és minél kevesebb lokális változóval** építsük fel a kódot. Ha van olyan lokális változó, aminek az értékét nem módosítjuk, akkor az újrastrukturálást igényel, vagyis paraméter lesz belőle.

- **Kerüljük az ideiglenes változók használatát**, mert növelik a komplexitást! Vagyis írjuk át az ilyen részeket, ahol lehet metódushívásokra!(Fowler szerint)

- Egy **attribútum hivatkozás helyett** mindig építsünk be egy **lekérdező metódust**! Általában a metódusok nagyon rövidek. Egy-két utasításból áll a törzsük, amik általában maguk is metódushívások. Metódushívások sorozatára kell bontani a kódot!

- Használjunk **beszélő neveket**.

- **Laza-, szoros kötődés.** A metódust azon osztállyal kell egységbe zárni, amelynek az objektumát manipulálja. Vagyis ha lehet **ne paraméterként, kapja meg a metódus az objektumot**, amin dolgozik. Ezután a metódushívást módosítani kell. Ekkor a kód újrastrukturálása módosítja a tervet.

- **Ne elágazó struktúrát használjunk**, hanem többszörös öröklődést a különböző kategóriák megkülönböztetésére!

Mikor kell refactoringot csinálni?

- **Ismételt kódrészletek:** Ha van egy osztály, és annak két különböző metódusában ismétlődő kódrészletek vannak, akkor abból a két metódusból egyet kell csinálni. Elképzelhető az, hogy ez a szituáció azért jött létre, mert ugyanazt a dolgot két különböző algoritmussal csinálták meg. Ekkor az egyiket ki kell dobni, és a másikat kell megtartani. Különböző osztályok metódusaiban is lehetnek ismétlődő kódrészletek. Ez lehet tervezési és implementációs hiba is. Meg kell vizsgálni, hogy ténylegesen melyik osztályhoz kell, hogy tartozzon ez a metódus, és abban kell megírni.

- **Hosszú metódus:** A metódusoknak nagyon rövideknek kell lenniük, a hosszú metódusokat szét kell választani, és több rövidet kell írni. (Pici megjegyzése: előbb-utóbb megtelhet a rendszerverem a sok metódushívás miatt, úgyhogy célszerű inkább megtalálni az arany középutat.)

- **Nagyméretű osztály:** Sok az attribútum, sok a metódus. Vagy az egységbe zárással lehet a baj, vagy tervezési probléma, vagy strukturált módon programoztuk le az osztályt. Megoldás az egységbezárás megvalósítása széttagolással, megfelelő felelősség hozzárendeléssel.

- **Metódusoknál sok a formális paraméter:** Nehezebben érthető, nehezebben használható. Sok paraméter esetén valószínűsíthető, hogy a metódus nem saját osztályon operál. A metódust át kell helyezni abba az osztályba, amelynek az objektumain operál.

- **Különböző változtatási igények:** Akkor áll elő, ha egy osztályt meg kell változtatni, és ez a változtatás másként történik, különböző szituációkban. Pl.: Ha az adatbázisban változás áll elő, vagy új jogszabály érkezik, másként kell változtatni az osztályon. Ekkor az

egységbezárással lehet a gond. Szét kell tagolni, mert más-más viselkedést mutat az osztály más-más szituációkban.

- **Sörét effektus:** Rendszerünkön kis változtatás is sok osztályt érint. Egységbezárási gond, tehát valószínűsíthető, hogy az osztályoknál a struktúra többszöröződik. Át kell alakítanunk a kódot ennek megfelelően.

- **Versengés**: Amikor egy metódus nem saját adatot használ, vagyis nem saját osztályának objektumain operál. Az osztályok versengenek egymás objektumaiért. Erre létezik egy külön tervezési minta a Visitor**.**

- **Adatcsoportosulás:** Akár attribútumok, akár formális paraméterek esetén jelenik meg. Azt kell megvizsgálni, hogy az adatcsoport egy elemét tudom-e úgy manipulálni, hogy a csoport többi elemét nem érinti. Ha igen, akkor ez a csoportosulás fölösleges.

- **Primitív típusok:** Ha olyan osztályt találunk, amely lényegében, funkcionalitásában megegyezik valamilyen beépített típussal, akkor ez nyílván fölösleges. Beépített típusú osztályt megírni fölösleges.

- **Többszörös elágaztatás:** Használjunk helyette polimorfizmust!

- **Párhuzamos öröklődési hierarchiák:** Az alkalmazás szempontjából több hierarchia van, és ua. az osztály megjelenik több hierarchiában.

- **Lusta (semmittevő) osztály:** Egy alkalmazásból azt az osztályt, amelyet nem hivatkozunk, el kell távolítani.

- **Spekulatív általánosítás:** Tipikus implementációs hiba, tipikus programozói hozzáállási hiba. A programban hagyom az osztályt, bár nem kell.

- **Átmeneti/Ideiglenes mezők:** Több hibalehetőség, nehezebben érthető a kód, ezért próbáljuk kihagyni, paraméterré, visszatérési értékké alakítani.

- **Üzenetláncok:** Amikor egy objektum egy másik objektumot hivatkozik. Valószínűleg implementációs hiba, félreértett tervezés.

- **Nem megfelelő bizalmasság:** Nem megfelelő bezárási szint. Amikor egy osztály túl sokat mutat meg az eszközeiből, olyanokat is, amik fölöslegesek.

- **Alternatív osztályok különböző interfésszel:** Ugyanolyan funkcionalitású metódusok szignatúrája, specifikációja különbözik. Különböző osztályokban, különböző szignatúrájú metódusok ugyanazt csinálják. Metódusokat össze kell vonni, esetleg átnevezni és egy osztályba gyűjteni.

- **Adatosztályok:** Csak attribútumaik vannak, esetleg beállító és lekérdező metódusaik, nincs viselkedésmódjuk.

- **Elutasított örökség:** A bizalmasság ellentétje. Rossz az öröklődési hierarchia. Amikor egy alosztály átveszi a viselkedésmódot, de nem támogatja a szuperosztály interfészét.

- Az alkalmazott API osztályai nem egészen úgy működnek, ahogy kell.

Fowler szerint egy újragyártott kódban fölöslegesek a megjegyzések.

Az újragyártásnak van katalógusa ezzel a felépítéssel:
- Név
- Összefoglaló rész - Mikor és mire használható.
- Motivácó - A felhasználás körülményeire vonatkozik.
- Működés - Lépésről lépésre megadja, hogy az adott újragyártásnál mit, hogyan kell csinálni.
- Alkalmazási példák

Verifikáció és validáció

A terv alapján implementáljuk a szoftvert. Szűkebb értelemben a verifikáció és a validáció a programra vonatkozik. Általánosabb értelemben a teljes rendszerfejlesztési folyamatban beszélünk **verifikáció**ról és **validáció**ról.
A **verifikáció** azt vizsgálja, hogy a szoftver (program, dokumentáció és adatok) megfelel-e a specifikációjának. Verifikálni mindent kell.
A **validáció** azt vizsgálja, hogy a felhasználó számára megfelelő-e a szoftver, vagyis a szoftver specifikációja. A hangsúly itt már a programon van. Általánosabban mindkettő valamilyen megfelelést vizsgál.

Mi befolyásolja a szoftver elfogadási szintjét?
- A **funkciója**, vagyis mire akarjuk használni. Pl. Pakson értelemszerűen magasabb.
- **Elvárás**. Manapság a szoftvertől nem várjuk el, hogy megfelelően működjön, mert erre kondícionált minket a szoftveripar.
- **Piac kényszerítő ereje** arra készteti a cégeket, hogy minél hamarabb kész legyen a szoftver -> rengeteg béta.

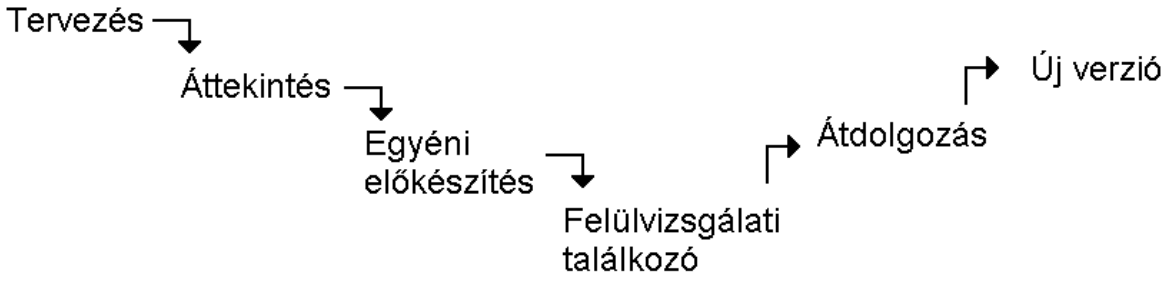
V&V technikák:
1. **Átvizsgálás**:
   Statikus technika. A szoftver valamilyen reprezentációját nézzük át, vizsgáljuk meg. Lehet ez dokumentáció, forráskód, specifikáció, bármi, amihez nem kell a program futtatása. Átvizsgálásnak egyik módja az emberi szemmel történő átvizsgálás. A másik az automatikus statikus elemzés: A program forrásszövegét nem ember, hanem program (egy intelligens fordító) vizsgálja át. Előkészíti az emberek feladatát, sok plusz információt szolgáltat.

   Átvizsgálás megközelítések:
   - **Vezérlés**: A felesleges, vagy nem használt kódrészletek felderíthetőek sokkal könnyebben, mint kézzel.
   - **Adathasználat**: A változók egymás utáni értékadásainak kiderítése. Feladata a nem típusos nyelveknél nagyobb szerepet játszik.
   - **Útvonalelemzés**
   - A **formális módszerek**: Formalizált tervekből generál kódot, vagy megvizsgálja, hogy a kód származtatható-e a specifikációból. Reprezentáció-transzformáció történik. Drága, időigényes. A formális módszerek csak igen nagy biztonságot igénylő rendszereknél fontosak. Alapja a matematikai logika.
     Formális módszertanok: B, SDL, BDM**,** Clean Room.
     Inkább eszközök: VDM, Z.

   Átvizsgálás előnyei a teszteléssel szemben:
   - A tesztelés nem derít fel hibacsoportokat.
   - A tesztelésnél a mellékhatások soha nem derülnek ki.
   - Tesztelést csak futtatható kódon lehet végezni.
   - Az átvizsgálás olyan verifikációs szempontokra is figyelemmel lehet, amikre tesztelésnél általában nem. Pl. Bizonyos szabványok betartása. A rossz programozási

stílus kiderítése. Kód megérthetőségének a vizsgálata.

Az átvizsgálását is csapat végzi, amelynek része a program írója. Először átnézik specifikációt közösen, és aztán egyedileg nézik át. Aztán összeülnek és megbeszélik. Megállapítják a specifikációnak nem megfelelést. Ezután a program írója megcsinálja a belövést. Majd megint összeülnek, és kezdik előről. Az átvizsgálást végzőknek járatosnak kell lenniük az adott nyelvben, az adott szakterületen.



Milyen jellegű hibákat tud felderíteni egy ilyen csapat?

- Adathibák. Minden változó kapott-e értéket.
- Tömb indexszelése megfelelő-e.
- Van-e puffertúlcsordulás.
- Vezérlési hibák.
- Elágaztató utasítások.
- Ciklusok szabályosan fejeződnek-e be.
- I/O hibák.
- Váratlan inputok esete.
- Interfészhibák.
- Paraméterkiértékelés, -átadás.
- Tárkezelés.
- Kivételkezelés.

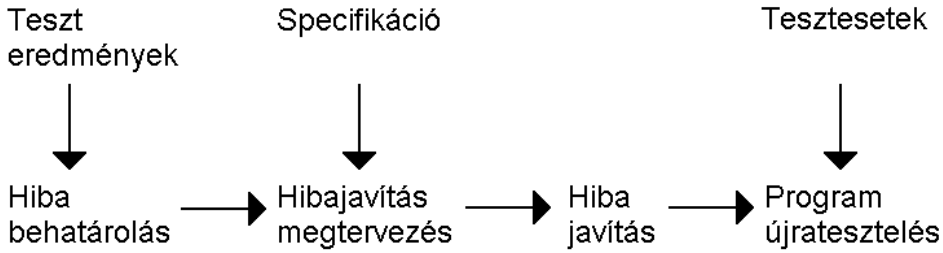
2. **Tesztelés**:
Dinamikus technika. A programot teszteljük futtatással. Nagyon ritkán végzi a program írója. Külön tesztelő csoportok vannak. A tesztelés nem bizonyít semmit. Pláne azt nem, hogy hibátlan a szoftver. A rendszerteszteket a legtapasztaltabb embereknek kellene végezniük, ezzel szemben sok cég friss diplomásokat alkalmaz tesztelésre.

Fajtái:

1. **Validációs**: A teszteléssel azt próbáljuk belátni, hogy a felhasználónak megfelelő-e a program. Azt próbáljuk bizonyítani, hogy a szoftver jó.
    a) **Statisztikai** - Metrikák lsd. később. Valamit mérünk, és abból próbálunk következtetéseket levonni.
    b) **Megbízhatósági** - A felhasználó számára történő megbízhatóság tesztelése.
2. **Hiányosság** - A programban lévő problémák felderítését szolgálja. Akkor sikeres, ha minél több hibát felderítünk.

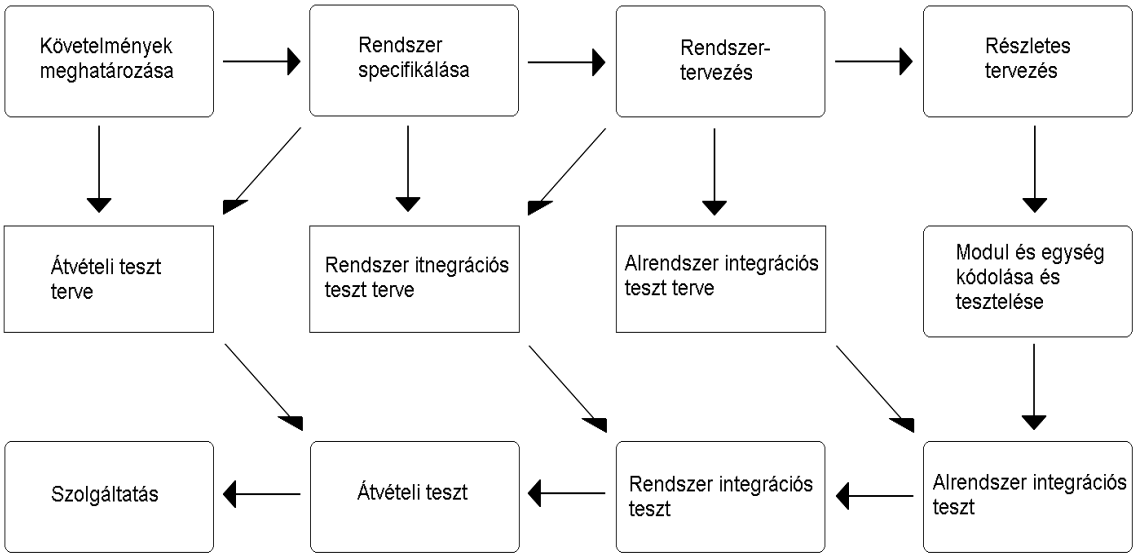
Belövés: Élesen szemben áll a teszteléssel. Belövéskor megpróbáljuk kijavítani a hibákat. Ami azzal kezdődik, hogy megpróbáljuk behatárolni a hiba helyét. A tesztelés csak felderíti a hibát. A debugger belövési eszköz éppen ezért. Mindig a program írójának a feladata.

Tesztelés, belövés:



Regressziós vagy újratesztelés. Újra lefuttatjuk a tesztet, ugyanazokra az input adatokra, amelyek korábban a hibát okozták. Ezzel bizonyítjuk, hogy a javítás sikeres volt.

A verifikáció és validáció az egész fejlesztési folyamaton végigvonul, nem az implementációnál kezdődik. Erre utal a klasszikus 3. V betű.
V&V elemei (a 3. V betű):



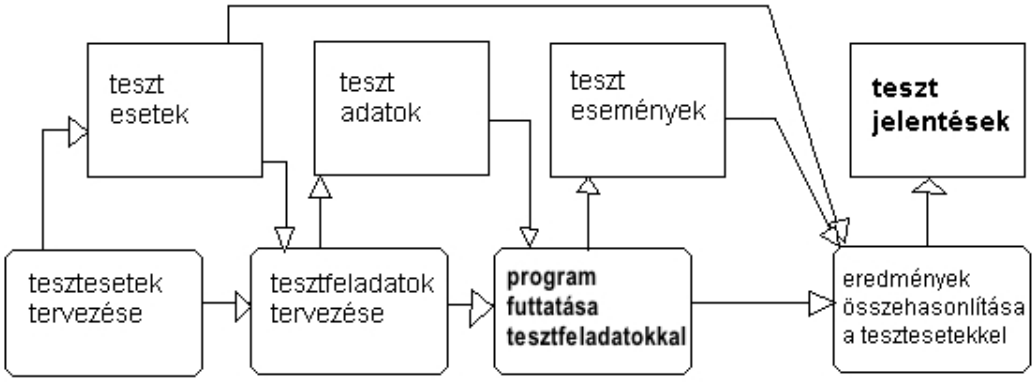
A nagyméretű információs rendszerek alrendszerekre bomlanak. A rendszerarchitektúra igazándiból alrendszerarchitektúra elsődlegesen.
Az alrendszerek modulokra bomlanak. Az alrendszer és modul között a következő a különbség. Az alrendszerek önállóak, megfelelő rendszeren belüli funkcionalitással.
Az alrendszerek megfelelő interfészeken keresztül kommunikálnak. Magyarán ha egy alrendszert kiemelek, attól még a rendszer megy tovább, max. a kivett funkcionalitás hiányzik belőle. A modulok nem önállóak. A modulok egymásra épülnek, egymás funkcionalitását használják.
A tesztelés leköveti a rendszer struktúráját. A tesztelést is tervezzük.
Mit kell tartalmaznia egy tesztterv?

- Le kell írni a tesztelési folyamatot magát.
- Le kell írni azt, hogy melyik követelmény tesztelése folyik az adott lépésben.
- Meg kell mondani, hogy a rendszer melyik részét teszteljük. Metódus, osztály stb.
- A tesztterv ben kell lennie ütemezésnek.
- Meg kell határozni, hogyan történik a dokumentálása a tesztnek.
- Meg kell adni a hardver és szoftverkövetelményeket.
- A tesztelésre vonatkozó korlátokat, megszorításokat.

A tesztterv nem statikus dokumentációk.

Az **agilis fejlesztési módszertan** nem választja el a tesztelést az egyéb lépésektől.

Modulteszt, egységteszt: A modul az a legkisebb egység, amit a programozó fejleszt ki. Pl. egy metódus vagy egy osztály, de osztálynál nem nagyobb. Ezeket a modulokat össze kell integrálni alrendszerré.
Az alrendszer **integrációs teszt** terve a rendszertervezésből jön. Az integrációs tesztet külön tesztelő csapatok végzik. Az alrendszerekből össze kell integrálni a rendszert.
Átvételi teszt, vagy átadási teszt: Amíg az eddigi tesztek tipikusan hiányosságtesztek, az átvételi teszt az validációs teszt. Az átvételi teszt demonstrálja a felhasználónak, hogy azt a rendszert kapja, amit várt. A követelményekből és a specifikációból jön.



Teszteset: Adott tesztben szereplő inputok és outputok együttese. Az outputtal van a probléma. A felhasználók ismereteit kell felhasználni, teszteseteket nem lehet automatikusan előállítani. A tesztesetet meg kell tervezni. Végigkíséri a teljes tervezési folyamatot.

Teszt adat: előállításuk vagy manuálisan, vagy automatikusan történik.

Kimerítő tesztelés: az összes elképzelhető végrehajtási szekvenciát leteszteljük. Ha van ciklus, a szekvenciák száma végtelen. (Ki kell alakítani tesztelési irányelveket!)
Ha menüvezérelt a program, akkor minden menüpontot tesztelni kell. A funkciókombinációkat is tesztelni kell. Azokat a szekvenciákat tesztelni kell, amikben az összes utasítás benne van. Az input adatokat is jól meg kell választani, helytelen inputra is tesztelni kell.
OO rendszerekben metódusokat, objektum különböző állapotait, az osztályt, framework-öket tesztelhetjük.

<u>Tesztelés fajtái:</u>

- **Rendszertesztelés** két fajtája:
    1. **Integrációs tesztelés:** Alrendszerek összeépítését teszteli. Tipikus hiányosság tesztelés. Regressziós teszten alapul. Történhet:
        a) **Lentről felfelé:**
        A modulok összeépítéséből kapjuk a rendszert.
        Teszt meghajtók kellenek, melyen a környezet viselkedését szimulálhatjuk. Akkor használjuk, ha az architekturális terv később alakul ki. Nehezebb a teljes rendszer működését demonstrálni.
        b) **Fentről lefelé:**
        Az egész rendszert teszteljük.
        A magas szintű elemek tesztelésével kell kezdeni.
        Bizonyos funkciók még nem implementáltak mikor tesztelünk, ezért szimulálni kell őket. A szimuláció időnként problémás.
    2. **Kiadási tesztelés:** Validációs tesztelés. Ha a felhasználót is bevonjuk a tesztelésbe,

akkor elfogadási tesztelés. Itt az ún. **fekete doboz teszt**et alkalmazzuk. Nem ismerjük a kódot, inputokat adunk neki és figyeljük az outputot.

Az inputokat hogyan tervezzük meg?

- A rendellenes inputokat vissza kell dobnia a rendszernek.
- Ki kell deríteni a puffertúlcsordulást okozó inputot.
- Ugyanazt az inputot többször is ki kell próbálni.
- Minden inputnak van egyfajta tartománya. A tartomány szélsőséges adataira is tesztelni kell.

A szoftvereknél egy új kiadás előállításánál a kiadási tesztet a cég végzi. És, ha egy adott szinten megbízhatónak minősítik a rendszert, akkor azt mondják, hogy előállt egy **alfa verzió**. Ekkor jön a **béta teszt**, amibe már külső embereket is bevonnak. Ezután előáll egy olyan kiadás, amiben a hibák számát lecsökkentették. Ezek után megjelenik a szoftver. Adott megbízhatósági szintű, senki sem állítja, hogy nincs benne hiba.

- **Stressztesztelés:**
  Teljesítmény teszt. A rendszert megpróbáljuk egyre nagyobb tesztelésnek kitenni. Két irányba mehet el**. Egyre nagyobb adatokkal tesztelünk, vagy a rendszerrel való interakciók számát növeljük**. A normális rendszer a stressztesztelés hatására is működőképes marad. Folyamán kimérhető a **működési profil**. Az a rendszer a jó, ami hibatűrő, nincs olyan inputja, ami kifekteti, hangolható. A működési profil a hangoláshoz fontos. Pl.: tartalmazza azt az információt, hogy a program mely funkciókat használja sokszor. Ezeknek a funkcióknak kell nagyon jól működniük.
  Ma problémát jelent, hogy a rendszerek kis része hangolható. Nagy rendszerek esetén kevés informatikus képes arra, hogy hangolja.

- **Interfésztesztelés:**
  Az OO rendszerekben felerősödött. (Itt interfész alatt a kommunikációs felületet értjük).
  Következő módon osztályozható:
  - **Paraméter interfész:** Ezeken keresztül adatok továbbítódnak. Néha függvény referenciák.
  - **Osztott memória interfész:** A modulok közös memóriaterületet használnak.
  - **Procedurális interfész:** Osztályok kommunikációs interfésze, vagy egy csomag interfésze.
  - **Üzenettovábbító interfész**: Üzenetek segítségével interfésztesztelés, mint kommunikációs teszt.

- **Komponenstesztelés, modultesztelés, egységtesztelés:**
  A fejlesztő végzi. A rendszer olyan kis részeire vonatkozik, melyek önállóan tesztelhetők. A belövési folyamat is egyszerűbb. A komponenstesztelés majdnem minden esetben hiányosság tesztelés is.
  Teszteset tervezés modultesztelésnél:
  1. **Követelmény alapú tesztelés:** A rendszerfejlesztés megkezdése előtt jelenik meg. Magas szintű tesztek, a validációs és hiányossági tesztelést egyaránt szolgálják

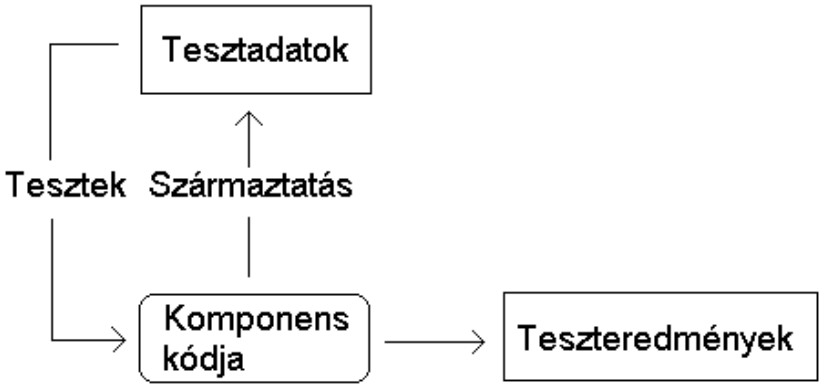
  2. **Partíciós tesztelés:** A fekete dobozos tesztelésnek egyfajta változata. Az input adatokon végrehajtunk egy úgynevezett **ekvivalencia osztályozást**. Partíciós tesztelésnél meghatározzuk az ekvivalencia osztályokat részben a specifikáció alapján, részben az adott szakterület ismeretében. Általában feltételezzük, hogy ha egy ekvivalencia osztályt felderítettünk, akkor ebbe az osztályba eső bármely adatra

a program ugyanúgy fog viselkedni, mint bármelyik másik ekvivalencia osztálybeli adatra.

Partíciós teszt ökölszabályok:

- Egy elemű sorozatot is tudni kell kezelni (nem csak több eleműt) amikor valamilyen sorozatot kezelünk. Gyakran egyelemű sorozatokat nem kezelik az alapszoftverek.
- Ugyanarra a tesztadat sorozatra többször is le kell futtatni a programot.
- Különböző méretű input adatra kell futtatni teszteket.
- Az ekvivalencia osztályba tartozó adatok, ha rendezettek, akkor olyan tesztadat sorozatokat kell választani, amiben az ekvivalencia osztály szélső elemei és közepén lévő elemek is benne vannak. És ezekre kell tesztelni a programot.

3. **Struktúra teszt (fehér doboz):**



A fekete doboz teszttel szemben itt ismerjük a kódot, és a kód ismeretében határozzuk meg a tesztesetet, a kód ismeretében generáljuk az adatokat. Tipikus modul teszt. Az adott modul írója ezzel a módszerrel tesztel. Integrációs tesztnél nem jöhet szóba.

A kód ismeretében újabb ekvivalencia osztályok hozhatók létre mondjuk a partíciós teszteléshez, újabb tesztesetek adhatók meg, mintha csak a specifikációt ismernénk.

4. **Útvonaltesztelés:** Tipikus modultesztelési eljárás. Tudjuk, hogy a kimerítő tesztelésnél az összes lehetséges végrehajtási szekvenciát teszteljük. E helyett jön az útvonal teszt, amikor különböző futási szekvenciákra tesztelek. A program vezérlési szerkezetében felfedezhető úton legalább egyszer végigmegyek. Ehhez megadhatjuk a program folyamatgráfját, ami a vezérlési szerkezetet tükrözi. Ha nincs goto, akkor ez meglehetősen gyorsan, és automatikusan megy.

Automatizált teszteszközöket használnak általában.

(ábra)

A dinamikus elemzőnek az a feladata, hogy futási profilt állítson elő.

…

Minőségkezelés

A szoftvernek, mint minden ipari terméknek van minőség aspektusa. Ez annyit jelent, hogy az adott termék megfelel a specifikációjának. A szoftver speciális termék, mert nem lehet gyártani, hanem szellemi termékként állítják elő. A szoftvert a követelmények alapján tervezik, de a probléma ott van, hogy a követelmények nem biztos, hogy pontosan felderíthetők, kideríthetők, elemezhetők, tervezhetők. Egy verifikált szoftver lehet a specifikációjának megfelelő, de nem biztos, hogy validált. Az ipari termékeknél egy verifikált szoftver általában validált, mert a specifikációja validált. A specifikációt még nem tudjuk validálni a szoftvernél.

Fogalmak a minőségkezelés vonatkozásában:

- **Minőségbiztosítás:** A jó minőségű szoftverek előállítását eredményező szabványok és előírások. A minőségbiztosítás egy szabványeggyüttes, vagy eljáráseggyüttes.
- **Minőségtervezés:** Amikor egy konkrét szoftverprojekt vonatkozásában, kiválasztjuk az előbbi szabvány és eljáráseggyüttesből azokat, amelyek illeszkednek, megfelelnek az adott projekthez. A projektre szabjuk a minőségbiztosítás által adott szabványokat, eljárásokat.
- **Minőség-ellenőrzés:** Azoknak a folyamatoknak a meghatározása és kötelező követése, amelyek biztosítják, hogy a projekt fejlesztőcsapata alkalmazza a minőségtervezésben meghatározott szabványokat és eljárásokat.

A minőségkezeléshez kell egy mindenkori szoftverfejlesztésektől független minőségkezelő, **minőségbiztosító csapat,** amely folyamatosan fejleszti, karbantartja a vállalati minőségbiztosítási szabványokat, eljárásokat.
Kell egy **minőségtervező csapat**, amelyben benne kell legyen a projektvezető.
A **minőség-ellenőrzés** pedig teljesen független a fejlesztőcsapattól.

Az ipari termékeknél a minőségbiztosítási szabványok egyértelműen folyamatszabványok. Klasszikus minőségkezelés azt mondja, hogy biztosítani kell, meg kell határozni a folyamatot, aminek eredményeként előáll a termék, és ha a folyamat jó a termék is jó lesz. Szoftvernél ez kicsit másképp van, nem elég csak a folyamatot kezelni, mert úgy még nem garantált a minőségi szoftver.

(ábra)

Szoftver minőségbiztosításnál a szabványok két nagy fajtájáról beszélünk:

1. **Folyamatszabványok**, amelyek a tevékenységhez kötődnek. Végigkövetik a szoftverfejlesztés teljes ciklusát.
2. **Termékszabványok** a folyamat végeredményeként előálló szoftvertermékre vonatkoznak.

A szoftverfejlesztésben a szabványok a következő 3 területen nyújtanak segítséget, útmutatást:

1. A termék és folyamatszabványokban összegyűjtjük a legjobb és a legrosszabb gyakorlatokat.
2. A szabványok jól definiált keretrendszert adnak. Ez a keretrendszer minden szoftverprojektre jól testreszabható. Keretrendszer, amelyben a projektet meg lehet szervezni a szabványok mentén.
3. Egy szoftverprojektben a szabványok biztosítják a folytonosságot. Ha valaki kiszáll a fejlesztésből, és szabványosan fejlesztették a projektet, akkor bárki be tud szállni.

Minőségkezelés lépései:

1. **Minőségbiztosítás**
   Minőségkezelés 1. lépése a minőségbiztosítás. Minőségbiztosítás alatt a jó minőségű szoftverek előállítását eredményező szabványokat és előírásokat értjük. A minőségbiztosítás egy szabványeggyüttes, vagy eljáráseggyüttes.

2. **Minőségtervezés**
   Minőségkezelés 2. lépése a minőségtervezés. A minőségkezelés szabványait az adott projektre szabjuk. Megtervezzük, hogy az adott fejlesztési projektben milyen eljárásokat, módszereket, szabványokat fogunk alkalmazni és hogyan. Ha a projekt új eljárásokat, módszereket, szabványokat kíván, akkor ezeket meg kell alkotni és beépíteni az eddigi rendszerbe.
   Minőségtervezésnél a következő kérdésekre kell kitérni (ökölszabályok):

- **Be kell mutatni a szoftvert**, abban az értelemben, hogy a szoftverrel szemben milyen minőségi elvárások vannak.
- **Meg kell tervezni a szoftver kibocsátást**. Mikor jelenik meg a szoftver, vagy mikor adjuk át, milyen kötelezettségeket vállalunk a szoftver működtetése idején, milyen követést vállalunk.
- **Meg kell tervezni a fejlesztési folyamatot**, vagyis hogy milyen minőségi szabványokat használjunk a fejlesztési folyamat közben.
- Meg kell mondani, hogy a szoftver minőségi jellemzői közül, melyeket tekintünk alapvetőnek. **Milyen minőségi célokat tűzünk a szoftver elé.**
- **Kockázatelemzési fázis**, vagyis a projekten belül milyen kockázatok merülhetnek fel, amelyek befolyásolhatják a minőséget.

Minőségi jellemzők (külső tulajdonságok, a nem funkcionális követelményeknek megfelelő jellemzők):

- **Biztonságosság**. A szoftver nem okoz kárt.
- **Biztonság**. A szoftver védett a szándékos vagy véletlen károkozással szemben (hack, crack ellen védett).
- **Megbízhatóság**. Ha lefuttatok egy scriptet, akkor lefut, és ugyanazt az eredményt adja ma is, holnap is, meg tegnap is:-)
- **Robosztusság**. A felhasználói hülyeségek ellen védett a rendszer. Hibatűrő rendszer.
- **Rugalmasság**
- **Érthetőség**
- **Tesztelhetőség**. A felhasználó oldali tesztelés. A nagyobb rendszerek rendszerverziókban működnek.
  Alapvetően 3 olyan rendszerverzió van, amit ma alapvetően, a nagyobb rendszerektől megkövetel a piac:
  - **Termelő rendszer vagy éles rendszer**: Ez az, ami folyamatosan működik.
  - **Fejlesztői rendszer**: Ebben az üzemeltető cég informatikusai dolgoznak, hangolnak, fejlesztenek.
  - **Tesztrendszer:** Ebben a felhasználók a felhasználói tesztesetek alapján generált tesztadatokkal tesztelik a rendszert.
  - **(Homokozó):** Teljes rendszer melyben pl. saját mintaalkalmazásokat, lehet kipróbálni mindenféle következmény nélkül.

  Az első 3 rendszer egymás mellett fut. Fejlesztői rendszerből a termelői rendszerbe csak akkor kerülhet át egy program, ha azt leteszteltük, és a tesztek azt mutatják, hogy az adott rendszer le van tesztelve egy adott mértékben. Csak tesztrendszerben tesztelünk (lásd ETR 1-es szórás esete)!
- **Alkalmazhatóság**. Az adott felhasználói körökben való felhasználhatóságot jelenti. Szokás testreszabhatóságnak is hívni. Ez általában azon szoftvereknél kerül elő, amelyek nem célrendszerek, hanem általános célra fejlesztik őket (pl. ETR).
- **Hordozhatóság**. Platformok között a rendszer átvihető-e.
- **Hatékonyság**
- **Tanulhatóság**
- **Újrafelhasználhatóság**
- **Karbantarthatóság**

Ezek a tulajdonságok egyszerre nem optimalizálhatók, eléggé ellentmondanak, egymást ronthatják. Ezért kell kiválasztani a fő tulajdonságokat, fő minőségi célokat, és a minőségtervezést arra optimalizálni. Pl. robosztusság és hatékonyság, egyszerre nem megy.

3. **Minőség-ellenőrzés**
   Minőségkezelés 3. lépése a minőségellenőrzés.

A minőségellenőrzésnek két közelítése van:

a) **Minőségi felülvizsgálat**. Programvizsgálat a minőség szempontjából. Emberek végzik, az adott szoftverreprezentációt tekintik át, és ellenőrzik, hogy a minőségtervezésnél testreszabott eljárások, szabványok betartásra kerültek-e. Külön minőségellenőrző csapatok vannak, amely csapatokba természetesen be kell vonni a projektvezetőt is.

b) **Automatikus szoftvermérés**. A szoftvermérés azzal foglalkozik, hogy hogyan lehet egy szoftver belső-, vagy külső jellemzőjéhez hozzárendelni egy numerikus értéket, hogy lehet megmérni a jellemzőt, és aztán hogyan lehet értelmezni ezt a numerikus értéket. Ezt az értéket nevezzük metrikának, szoftvermetrikának.

A szoftvermérés alapvetően két dologra használható:

1. **Általános előrejelzést** tudunk adni segítségével a szoftverről. Fontos lehet az erőforrásigények előrejelzése.

2. A szoftver **rendellenesen működő komponenseit kiszűrjük** a segítségével.

Amikor szoftvermérést végzünk, és előállítunk metrikákat, akkor ezekkel a következőket tehetjük meg:

- **Tárolhatjuk adatbázisban**, és ekkor egy új mérést összehasonlíthatunk az adatbázisban tárolt adatokkal. Ez alapján becsülhetünk, jelezhetünk előre bizonyos dolgokat, vagy ez alapján mondhatjuk azt, hogy valami rendellenesség van, mert a mért érték nagyon eltér az előzőektől. Az 1. esetben a hasonlóságot használjuk fel, vagyis mértem valamit és most is hasonlót, tehát később is így fog menni. A 2. esetben pont az ellenkezője következik be.
- Bizonyos metrikákra lehetnek szabványok, és ekkor a mért értéket **a szabványhoz hasonlítjuk**, és ez alapján tudunk valamire következtetni.

A belső jellemzők jól mérhetők. Ezekből mindenféle metrika jól számolható, származtatható. Külső jellemzők többnyire nem mérhetők, tehát metrika belőlük közvetlenül nem számolható, ezért a külső jellemzőket megpróbálják visszavezetni valamilyen belső jellemzőre, vagyis belső jellemzőhöz tartozó metrikából próbálnak következtetni a külső jellemzőkre.
Egy ábra a külső-, belső jellemzők kapcsolatáról.
(ábra)

Megjegyzés:
Látható, hogy a kézikönyv hossza nagyszerűen mérhető mondjuk karakterek számával, és ebből valamilyen módon következik a használhatóság, érthetőség. Viszont a hordozhatóságra hiába mondom, hogy 21.

Metrikák

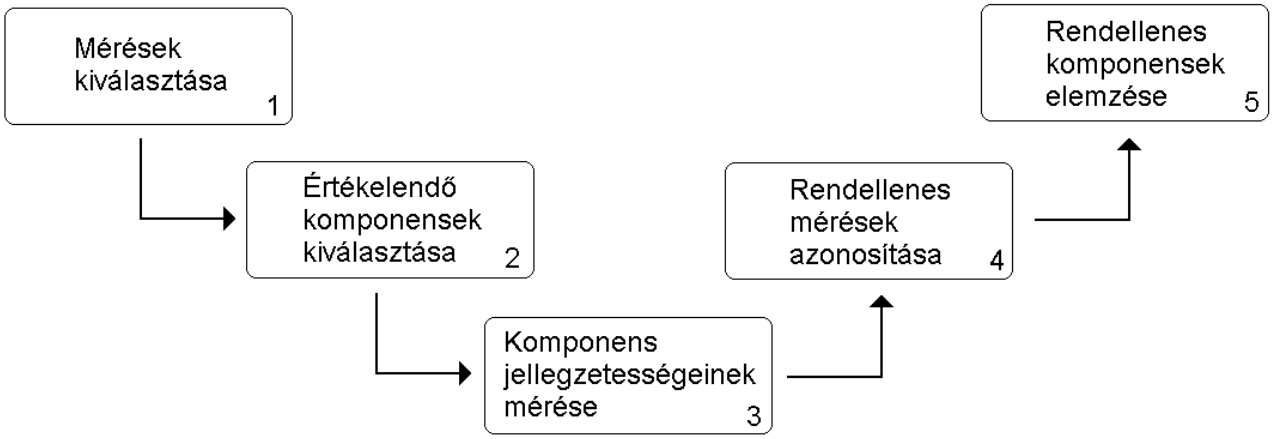
Metrika osztályok:

1. **Termékmetrika** - A szoftverre, mint termékre vonatkozik. A termékmetrikákon belül:
   a) **Dinamikus metrika** - A programhoz köthetők, és csak a futás közben állítható elő, mérhető meg.
   b) **Statikus metrika** - A szoftver bármely elemére vonatkozik (doksi, forrás stb.), és nem kell futatni, mivel nem is lehet. Általában könnyen számolhatók.
2. **Folyamatmetrika** - A szoftver előállításának életciklusára, mint folyamatra vonatkozik.

Az 1, 2 is lehet:

a) **Primitív metrika** – Az, ami közvetlenül számolható valamilyen jellemző alapján.
b) **Származtatott metrika** - Nyílván más metrikákból számolható. Általában sokkal jobban számolhatók, mint a primitív metrikák, talán azért is, mert gyakran dinamikus és statikus metrikák együtteséből származik.

Szoftvermérés-folyamata:



1. Meg kell fogalmazni a szoftverrel kapcsolatos kérdéseket! (pl.: Miért nem tudom felvenni a tárgyamat 3. napja ETR-ben?) Ki kell választani azokat a méréseket, amelyek a kérdéseket megválaszolják!
2. Ki kell választani azokat a komponenseket, amelyeken a mérést végezzük!
3. Végrehajtjuk a mérést. Előállítjuk a metrikákat.
4. A mért értékeket szabványokhoz, vagy az adatbázisban tárolt értékekhez hasonlítom. Relatív értékek vannak, vagyis a szokásoshoz közeli vagy kiugró értéket mértem-e.
5. Miért mértem kiugró értéket? Itt jön a magyarázat. A mért értékek mit jelentek. Ehhez vissza kell nyúlni a komponenshez. A komponens segítségével próbálom megmagyarázni. Ez lehet egy iteratív folyamat, mert a végeredmények segítségével pl.: pontosítani tudom a kérdéseket.

Egy ideális metrika a következő 5 tulajdonsággal rendelkezik:

1. **Pontosan definiált**. Mindenki számára világos, hogy az adott metrika mit jelent.
2. **Objektív**. A mérést sokszor meg tudjuk ismételni, és mindenki végre tudja hajtani.
3. **Validált.** Azt méri, amit kell. A mérés definiálása után a mérést elég sokszor végrehajtva, a metrika valóban azt méri, amit kell.
4. **Robosztus**. A metrika viszonylag érzéketlen a folyamat, vagy a termék olyan változásaival szemben, amelyek nem a validált méréshez tartoznak. Amit nem mérek, az nem befolyásolja a mérést.
5. **Könnyen és olcsón végrehajtható** a mérés.

Termékmetrikák közül néhány:

- **Méretre vonatkozó metrikák** - A szoftver kiterjedését számszerűsítik valamilyen módon.
  - **Azonosítók hossza** - Automatikusan meghatározható. Egy hosszabb azonosító valószínűleg több információt árul el az adott eszközről, mint egy rövidebb (beszélő nevek). Viszont ha az átlagos hossz nagyon nagy, akkor érthetetlenné válhat a kód. Egyszerűen számolható, de nem sok mindenre jó.
  - **LOC - (Lines of Code) Kódsorok száma**. Problémája, hogy mit értünk alatta. Leggyakrabban használt az effektív LOC, ami a kód azon sorait veszi figyelembe, amelyek nem üresek, és nem megjegyzések.
    Egyszerűen számolható, de adott nyelvre, adott programozóra vonatkozik. LOC-ból olyan külső jellemzőkre következtethetünk, mint karbantarthatóság, érthetőség, tanulhatóság, hibák száma.

    - **FP - (Function Points).** A felhasználói inputok, az outputok, a lekérdezések, és a program által használt törzsállományok súlyozott együttes számát jelenti. Statikus**.** Olyan metrika, amelyet a projektre kell szabni. Az adott projekten belül kell megmondani milyenek a súlyok. Pl. input, lekérdezés 4, output 5, törzsállomány 10. Az így meghatározott számot még korrigálják. A program implementálása előtt meghatározható az értéke, a tervezés szakaszában meghatározható. Általában erőforrás-előrejelzésre használják.
- **Komplexitási metrikák** - A program összetettségét, bonyolultságát mérik. Statikus, belső mértékek. Minél nagyobb az értéke, annál nehezebben érthető, nehezebben újrafelhasználható, karbantartható, nagyobb valószínűséggel hibás a kód.
    - **Feltételes szerkezetek mélysége**.
    - **Ciklomatikus komplexitás** - Az útvonaltesztelésnél emlegetett folyamatgráf alapján méri a komplexitást. Ha "goto" mentes a program, akkor a következőképpen számolható: az élek számából levonjuk a csomópontok számát, és hozzáadunk kettőt. Igaz ez akkor, ha nem adatvezérelt a program, mert ebben az esetben nem biztos, hogy ér bármit is.
    - **Információáram** - Alprogramok komplexitását méri. A program működési komplexitását méri. Ennek segítségével a program futása közben felépülő hívási lánc mérete alapján állítunk elő egy metrikát.
        $C = [\text{alprg. hossza}] * b^2 * k^2$
        b: azon egyéb alprogramk száma, melyek az adott alprogramot meghívják.
        k: azon alprogramok száma, melyet az adott alprogram hív.
    - **Halstead-féle metrikák:**
        a) **n: Programszótár metrika** - Megszámolja a különböző operátorokat, és operandusokat, és ezeket összeadja. A számítás komplexitását méri.
        b) **N: Hosszmetrika** - Az összes operátort, és operandust összeszámolja, és összeadja. Ez egy méretmetrika.
        c) **Kötetmetrika** - A fenntiekből származtatja Halstead. A program futtatása során szükséges tárterületet becsli. $V = N * \log_2(n)$
- **Hiányossági metrikák:**
    - Terv változatok száma.
    - A szoftverátvizsgálások során talált hibák száma.
    - A tesztelés során adott időegység alatt észlelt hibák száma.
    - Kódmódosítások száma.
    - Megbízhatósági metrikák (dinamikusak)
    - **POFOD** - (Probability Failure On Demand) Annak a valószínűsége, hogy a rendszer adott szolgáltatáskérés esetén hibázik.
    - **ROCOF** - (Rate Of Failure Occurence) Hibaintenzitás. Adott időegység alatti hiba-előfordulások gyakoriságának a száma.
    - **MTTF** - (Mean Time Of Failure) Két hiba-előfordulás között eltelt átlagos idő.
    - AVAIL - (Availability) Rendelkezésreállás. Milyen valószínűséggel érhető el a rendszer.
- **OO metrikák közül néhány:**
    - **Number Of Attributes per class (NOA)**: Osztályok belüli attributumok száma
    - **Number Of Methods (NOM)**: Osztályok belüli metódusok száma.
    - **Message Passing Coupling (MPC)**: Üzenettovábbítás szorossága. Van *A B* osztály... Az *A* osztály két különböző metódusa meghívja a *B* ugyanazon metódusát. Megszámoljuk, hogy hányszor van ilyen szituáció. *A* különböző metódusai hányszor veszik igénybe ugyanazt a szolgáltatást *B*-től. Túl nagy érték nem jó, valószínűleg tervezési hiba.
    - **Data Abstraction Coupling (DAC)**: A megadott osztály által definiált absztrakt adattípusokat számlálja.
    - **Lack of Cohesion in Methods (LCOM)**: A metódusok kohéziójának a hiánya. A

metódusok eltérőségét méri az általuk használt példányszintű attribútumokon keresztül.

$m_1, m_2, \ldots, m_n$ az osztály metódusai,

$\{Ij\}$ az mj metódusai által használt példányszintű attribútumok halmaza.

Ekkor legyen

$P = \{(Ii,Ij) | Ii \cap Ij = 0\}\ i \neq j,$

$Q = \{(Ii,Ij) |\ Ii \cap Ij \neq 0\}.$

$LCOM = |P|-|Q|$ *ha* $|P|>|Q|,$

0 egyébként.

Akkor a jó, ha az értéke 0. Ha nem 0, akkor a funkcionalitással probléma van. Refactoring kell.

- **Loose Class Cohension (LCC)**: Laza osztály kohézió. Az osztály publikus metódusai közül azon metóduspárok százaléka az összes lehetséges metóduspáron belül, amelyek közvetlenül, vagy közvetett módon azonos attribútumokat használnak.

Folyamatmetrikák közül néhány:

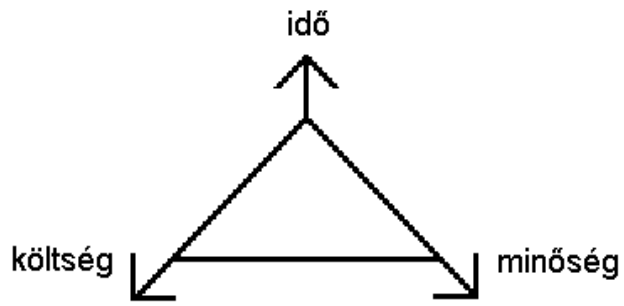
- Időmetrikák
- Erőforrások (munkaerő, pénz, hardver/szoftver)
- Bizonyos események bekövetkezési számát figyelem
- Tevékenység (ovális)
- Részfolyamat (ovális akármi beárnyékolt résszel)
- Résztermék (árnyékos téglalap)
- Feltételek, korlátozások, megszorítások (téglalap)
- Kivételek (dupla falú téglalap)
- Szerepkörök (árnyékolt kör)
- Folyamat iránya (nyilak) /balról jobbra/

Informatikai Projekt

A projekt egy olyan tevékenységsorozat, amelynek a főbb tulajdonságai a következők:

- Van valami meghatározott **konkrét cél**ja.
- A projekt **előállít valamit**.
- Van **kezdete és** van **vége**.
- Van **költségvetése**.
- **Erőforrások**at használ. (hardver, szoftver, humán-erőforrás)
- Vannak **minőségi kritériumok** a projekthez és a termékhez.

Projekt háromszög:



Ezek egymás ellen hatnak.

Minőség: Az elért terméket hasonlítják az elvárásokhoz. Ha 1 körül van a hányados, akkor jó volt a projekt. Ha 1-nél kisebb, akkor az elvárásokat nem sikerült teljesíteni. (Ha 1-nél nagyobb, az sem jó, mert elkényeztettük a felhasználót. by Pici)

Az informatikai projektek jelentős része most is megbukik. (kb. 20%-uk) Ami annyit jelent, hogy a projekt nem jutott sehova. Kb. 30% körül van a sikeres informatikai projektek száma. A maradék projektekben valami probléma lépett fel.

Ezek a problémák jelentkezhetnek az informatikai projekteknél:

- Nincs világosan megfogalmazva a cél.
- Kevés az erőforrás.
- Rossz a választott technológia.
- Gyenge a kommunikáció a projekt résztvevői között.
- Nincs követelmény-, és változtatáskezelés.
- Nincs normális dokumentálás.
- Gyengék az emberek.
- Vagy egyszerűen csak összevesztek.

Ha nincsenek ezek a problémák, akkor nagy valószínűséggel sikeresek a projektek.

Három fő ok, ami miatt megbukik egy informatikai projekt:

- Idő nem reális.
- A humán-erőforrás kezelésben probléma van.
- A projekt terjedelme nem megfelelő. (scope)

Az informatikai projektek specialitásai:

- Ezek a projektek általában **kicsik**.
- Az informatikai projektekben tipikusan okos, érzékeny, individualista emberek vesznek részt. **Magasan kvalifikáltak a résztvevők**.
- Általában tiszta informatikai projekt nincs; **részei más projekteknek**. Sok mindentől függ a projekt.

Informatikai projektek fajtái:

- **Termékfejlesztési projekt**: Valamilyen dobozos szoftvert állítunk elő. A fejlesztő cég egy informatikai terméket akar kifejleszteni.
- **Alkalmazásfejlesztési projekt**: Megrendelésre készül. Célszoftvert fejlesztünk. Van egy megrendelő, akinek az igényeit ki kell elégíteni.
- **Rendszerintegrálási projekt**: Meglévő rendszerek összepakolását végzi.
- **Bevezetési projekt**: Vagy lecseréljük az eddigi rendszerünket, vagy eddig nem volt és bevezetünk egyet.
- **Infrastruktúrafejlesztési projekt**: Hardvert és alapszoftvereket veszünk.
- **Tanulmánykészítési projekt**: Valamilyen környezetvizsgálat, felmérés, hatásvizsgálat a cél. Vagy magunk végezzük, vagy megrendeljük.
- **Tesztelési projekt**: Valamilyen rendszer tesztelését végzi.

Projekt folyamatai:

A projekt folyamatok több fázist, vagy az egész projektet végigkísérik. Az egyik legfontosabb folyamat, amivel indul az egész: projektbecslés, amiből meg kell mondanunk az időt és a költségeket. Tehát a projekt elején ütemezni kell. Nincsenek egzakt módszerek, csak becslésen alapuló módszerek.

- **Projektbecslés:**
  Következő idő -, és költségbecslési technikák alakultak ki:

a. **Metrikákon alapuló becslés**: Veszünk valamilyen termékmetrikát, és erről próbálunk meg valamilyen becslést adni az időre és a költségekre. COCOMO költségbecslő modell a leghíresebb költségbecslő modell:

$$ráfordítás = a * (méret)^b$$

Az a és b olyan állandó, amik gyakorlati projektek alapján lettek becsülve.
Ráfordítás: Az idő emberi hónapban.
Méret: Valamelyik méretmetrika. Attól függően, hogy a vizsgált projekteknek milyen jellemzői voltak (termék mérete, fejlesztő csapat mérete, határidő szorossága stb.). 3 kategóriát állapítottak meg;
egyszerű -> bonyolult (a=2,4..3,6; b=1...1,3 körül).
Ha a teljes átfutási időt is becsülni akarjuk (a teljes projekt idő hónapban):

$$2.5*(ráfordítás)^c$$ (*C : COCOMO még ezt hozzárakja)

Tehát a teljes projektidő stabilabb. Ebben a környezetben lehet figyelembe venni a projekt környezetét.

b. **Terven alapuló becslés**: A tervezési fázisban előálló terv egyes elemeihez egyenként rendelünk időt és költséget megfelelő módszerrel végrehajtunk egy ütemezést, ha összeadjuk, megkapunk egy teljes időráfordítást és teljes költségráfordítást.

- **Láthatási időn alapuló becslés**

a. **Terméken alapuló becslés**: A szoftvernek bizonyos jellemzőit hamar le kell és le lehet rögzíteni. (A ma divatban lévő interfész alapú terveknél a kifejlesztendő szoftver képernyőit hamar rögzíteni kell.) A részre leadott becslés nem biztos, hogy jól becsli a teljeset.

b. **Átfutási időn alapuló becslés**: Úgy becsül, hogy megvizsgálja egy adott időintervallumban az adott projektlépést mekkora és milyen minőségű csapat tudja végrehajtani? Az időt rögzítem, és az időhöz próbálom hozzárendelni a csapatot, és a költséget ebből pakolom össze.

c. **Analógiás becslés**: Az eddigi lefutott projektek között keresünk olyan projektet, ami a legjobban hasonlít a mostani projekthez és ismervén a lefutott projektnek az idő és költség igényét, és ebből próbálok becsülni -> a projektek adatait tárolni kell.

Ma mindenféle adatbázisok rendelkezésre állnak, mindenféle elemzést végrehajthatok, különös tekintettel az analógiás elemzésre. (A komoly projektek analógiás becsléseket használnak.)

- **Nyerő ár**: Melyik az az ár, amelyiken megnyerem a projektet? Az kapja a projektet, aki a legolcsóbban meg tudja csinálni. Veszélyes, mert abból kell kifejleszteni, de kevés pénz még mindig több, mint a semmi.

- **Design-&-schedule**: Adva van a termék kifejlesztési határideje. Innentől kezdve visszafelé tervezek. Időcsúszás nincs.

- Van x emberem, a felhasználó y idő alatt akarja a programot, a becslésem x*y.

Scope:

Arra vonatkozik, hogy meghatározzuk a projekt mivel foglalkozik és mivel nem. Ennek

során meghatározzuk a projektkövetelményeket, és ezek adják a projekt scope-ját.

Projektkövetelmények:

- Stratégiai.
- Technikai/technológiai.
- Más rendszerekkel való kapcsolatra vonatkozó.
- Logisztikai.
- Minőségi.

Követelménymenedzsment**:** Kezeli a követelményeket. Az egész projektmenedzsment kulcsa a változtatáskezelés. Mivel változnak a termékkövetelmények, változnak a projektkövetelmények, változnak a csapat tagjai, változnak a felhasználói igények.
A változtatást mindig dokumentálni kell, és a változtatást mindig engedélyezni kell. A nem engedélyezett változtatást nem lehet végrehajtani. Változtatáskezelési szabványok és technikák vannak. Ezek általában belső szabványok, mindenfajta dokumentumokat kell kitölteni sokszor. De ezt végig kell csinálni. A változtatáskezelésnek vannak automatizált szoftverei (a projekt vonatkozásában).

Konfigurációkezelés:

A projekt rendelkezésére álló hardver-, és szoftverforrásoknak a kezelése.

Kockázatkezelés:

Kockázat alatt egy olyan eseményt értünk, amely nem tervezett, de valamilyen valószínűséggel a jövőben bekövetkezhet. Fel kell készülnünk egy ilyen eseményre, meg kell becsülnünk, hogy milyen események milyen valószínűséggel és mikor következhetnek be. A kockázat nem biztos, hogy negatív (zömében az).

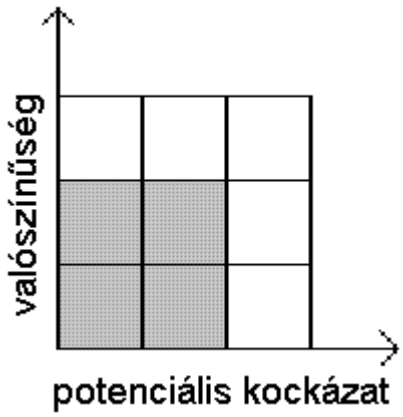
Kockázat kategóriák:

- Technológiai, módszertani.
- Pénzügyi.
- Kereskedelmi.
- Szervezeti.
- Humán-erőforrás kockázatok.

Kulcs-kockázatok:

- A projekt mérete növekszik (mert a felhasználó igényei nőnek, az elvárások nőnek). Ezzel számolni kell.
- Rossz volt a becslés.
- A csapatból távoznak alapvető szerepeket betöltő munkatársak.
- A csapat inhomogén. A termelékenység nagyon szór. Programozók hatékonysága nagyon eltérő lehet. Csapatproblémákat vethet fel.
- Alkalmazásfejlesztéseknél a megrendelő, és a szállító összeveszik.

Kockázatkezelés felderíteni a kockázatokat! Megbecsülni, hogy milyen valószínűséggel következnek be! Megbecsülni, hogy milyen károkat okozhatnak a kockázatok!
Kockázati mátrix: 3*3



Megjegyzés:
A szürkékkel nem foglalkozunk.
Valószínűség lehet kicsi, közepes, nagy. Minden kockázat vonatkozásában elhelyezzük a mátrixban.

Informatikai projektek dokumentációja, jogi vonatkozások

Megrendelő <- Fejlesztő

Fogalmak:

- **Ajánlat**: az ajánlat elfogadása a megrendelő részéről szerződésnek minősül. A magyar jogrend szerint szóban is lehet szerződést kötni.
  Egy ajánlatnak a következőket kell tartalmaznia:
  - Meg kell nevezni, hogy ki teszi az ajánlatot.
  - Le kell írni, hogy mi az ajánlat
  - Le kell írni azokat a feltételeket, amelyek mellett az ajánlat érvényes (határidő, mennyiért?)
  - Le kell írni, hogy meddig érvényes az ajánlat
- **Szerződés**: meghatározza azokat a kereteket, melyek mentén a projekt szerveződik.
  Szerződés fajták:
  - **Munka-szerződés**: A cég embert alkalmaz. Munkaviszonynak minősül.
  - **Vállalkozási-szerződés**: Jól körülhatárolt, konkrét feladatra vonatkozó szerződés. Vállalkozási díjjal egyenlíti ki a megrendelő.
  - **Megbízási szerződés**: Nem egy konkrét cél, hanem valamilyen tevékenység ellátását célozza meg. Általában valamilyen módon mérjük a befektetett munka mennyiségét és valamilyen egységekben fizet a megrendelő.
  - **Kutatási szerződés**: A vállalkozó semmilyen garanciát nem vállal.
  - **Licensz szerződés**: Valamilyen informatika szolgáltatást céloz. Ma meglehetősen divatos.

  Az ajánlatot és a szerződést is cégszerűen kell aláírni. Minden cégnek megvan, hogy milyen a cégszerű aláírás.
  A szerződés általában tartalmaz határidőket. Adott határidőre adott résztermékeknek el kell készülnie.

- **Projekt definíciós dokumentumnak**, amely pontosan leírja magát a projektet és mellette szokás egy minőségkezelési dokumentumnak is lennie. Ezek nélkül projekt nem indítható.

- **Szakmai dokumentum**: kimondottan az informatikai oldalról lényeges. Műszaki dokumentumok.

- **Projekt dokumentumok**: A projekt folyamán mindenféleképpen van naplózás, vannak emlékeztetők és jegyzőkönyvek. Emlékeztető minden meetingről. Jegyzőkönyv minden döntésről.

- **Teljesítésigazolások**: annak az igazolása, hogy a szerző eleget tett a szerződésben leírtaknak.

- **Záró dokumentum**: megállapítjuk, hogy sikerült-e, vagy nem leszállítani a projektet.

- **Projekt iroda**: Ennek a szervezetnek a feladata a vállalati projekt-szabványoknak a kidolgozása. Projekt folyamatok kezelése, modellezése, projekt-irányítási eszközök kidolgozása stb.

Jogi formulák:

- **Jótállás** (garancia): A garanciát adott időre nyújtja a termék, vagy a szolgáltatás előállítója. Az adott időn belül vállal garanciát. És, ha garanciális probléma van, akkor az előállítónak kell bizonyítania, hogy a felhasználó rosszul használta a terméket és ezért adódtak problémák. Az előállítónak kell orvosolnia a problémát.
- **Szavatosság**: A termékben nincs rejtett hiba. A szavatosság nem időkorlátozott dolog. Annyit jelent, hogy validált és verifikált a rendszerünk. A felhasználónak kell bizonyítania, hogy nem megfelelő a szolgáltatás.
- **Felmondás**: A felmondás időpontjában érvényes, nincs visszamenőleges hatálya. Bárki felmondhatja a szerződést. Az adott pillanatban a felek elszámolnak egymással. Egy idő előtti szeződésbontás.
- **Elállás**: Vissza kell állítani a szerződés előtti állapotot.
- **Előleg**: elő-finanszírozás. A megrendelő fizeti előre. A szerződés nemteljesítése esetén az előleg visszafizetendő.
- **Foglaló**: Egy mellék-kötelezettség. Biztosítandó a projekt végrehajtását. A megrendelő fizeti és, ha a projekt, vagy a szerződés a megrendelő hibájából teljesül, akkor a foglalót elveszti. Ha viszont a vállalkozó hibájából nem teljesül, akkor a foglaló kétszeresét kell visszafizetnie.
- **Bánatpénz**: az elálláshoz kapcsolódik. Annak az ára, hogy elállhasson a szerződéstől a két fél. Ha valamelyik fél eláll, akkor azt a pénzt fizeti a másiknak.

Szoftverfejlesztési folyamat

A szoftverfejlesztési folyamat jellemzői:

- **Érthetőség**: Mennyire pontosan definiált a folyamat, és az mennyire könnyen fogható fel a definíció alapján.
- **Láthatóság**: Mennyire látható kívülről, hogy a folyamat egyes tevékenységei mit eredményeznek.
- **Támogatottság**: Mennyire vannak automatikus eszközök a folyamat egyes tevékenységeihez?
- **Elfogadhatóság**: A projekt egyes szereplői számára mennyire elfogadható, használható a folyamat, vagy a folyamat tevékenységei.
- **Megbízhatóság**
- **Stabilitás**
- **Karbantarthatóság**
- **Sebesség** (Abban az értelemben, hogy az adott tevékenység alapján milyen gyorsan lehet leszállítani a szoftverterméket)

A szoftverfejlesztési folyamat továbbfejlesztése:

mérés->elemzés->változtatás->mérés...

Mérni az aktuális projektet mérjük. Lehet, hogy nem csak folyamatot mérünk, hanem terméket is. Az elemzés fázisban kiderítjük a gyengeségeket, (elképzelhető, hogy folyamatmodellt alkalmazunk, adaptálunk, vagy testreszabunk). A változtatás fázisában ezeket a gyengeségeket küszöböljük ki.

Egy folyamat megváltoztatásához a következők kellenek:

- Azonosítani kell valamilyen módon a második lépésben (elemzés), hogy mit akarunk megváltoztatni a folyamaton belül.
- Több változtatás is kiderül, sorrendbe kell állítani őket.
- El kell dönteni, hogy melyek azok, amelyek nagyobb minőségjavulást biztosítanak.
- Bevezetjük a változtatásokat.
- Majd jön egy betanítási lépés, mert meg kell szokni az új folyamatmodell alapján történő munkát.

A termék minőségére a következő 4 dolog van alapvetően hatással:

- Technológia.
- Emberek.
- Folyamat.
- A projekt olyan dolgai, mint költség és idő.

Alapvetően befolyásolja, hogy e 4 dolog közül melyik a lényeges, hogy mekkora méretű a projekt. Kis projekteknél a technológia és az emberek az alapvetők. Nagy projekteknél a másik kettő a lényeges. A nagy projekten sok ember dolgozik és az emberek minősége helyett sokkal inkább lényeges az, hogy legyen egy folyamatmodell, amit követni lehet. És sokkal lényegesebb a projektnek a megszervezése, alapvető a kommunikáció.